UNIVERSIDADE FEDERAL DE SANTA CATARINA
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

UMA PROPOSTA BASEADA EM PADRÕES DE DESIGN PARA O DESENVOLVIMENTO DE SISTEMAS COOPERATIVOS EM AMBIENTE ABERTO

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação

CEFERINO CASTRO CASTRO

Florianópolis, 21 de dezembro de 1999

UMA PROPOSTA BASEADA EM PADRÕES DE DESIGN PARA O DESENVOLVIMENTO DE SISTEMAS COOPERATIVOS EM AMBIENTE ABERTO

Ceferino Castro Castro

'Esta Dissertação foi julgada adequada para obtenção do Título de Mestre em Ciência da Computação, Área de Concentração *Sistemas de Computação*, e aprovada em sua forma final pelo Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina'.

Prof. Rosvelter João Coelho da Costa

Orientador

Prof. Fernando Álvaro Ostuni Gauthier

Coordenador do Curso de Pós-Graduação em Ciência da Computação

Banca Examinadora:

Prof. João Bosco Mangueira Sobral

Prof. Murilo Silva de Camargo

Prof. Roberto Willrich

Prof. Rosvelter João Coelho da Costa

AGRADECIMENTOS

Ao curso de pós-graduação em Ciência da Computação e a Universidade Federal de Santa Catarina pela infra-estrutura e organização que viabilizaram o desenvolvimento deste trabalho.

Aos professores João Bosco Mangueira Sobral, Murilo Silva de Camargo, Roberto Willrich e Rosvelter Coelho da Costa, por terem julgado este trabalho.

Ao meu orientador, Prof. Rosvelter Coelho da Costa, pela dedicação e pelos valiosos ensinamentos.

Aos meus pais, à minha esposa Carmen e aos meus filhos Marco, Robinson e Daniel pela paciência, carinho, compreensão e apoio constante.

RESUMO

Muitas ferramentas, técnicas e metodologias têm sido desenvolvidas para abordar problemáticas próprias do projeto de software orientado a objeto. Argumenta-se que os padrões de design podem ser utilizados como uma ferramenta essencial no projeto de software altamente reutilizável.

No trabalho apresentado nesta dissertação, uma questão importante sobre os padrões de design é abordada:

Os padrões de design constituem uma boa ferramenta para a construção de sistemas cooperativos flexíveis e reutilizáveis?

Para tanto, foi desenvolvido um sistema de conversação virtual sobre o ambiente Internet utilizando-se uma metodologia totalmente baseada em padrões de design. O sistema inteiro foi desenvolvido sobre a plataforma Java da Sun Microsystems. O nível de reutilização do sistema foi avaliado sob o aspecto funcional, estendendo-o com novas funcionalidades, e estrutural, modificando sua estrutura de interação.

Os padrões de design utilizados neste trabalho são os seguintes: Abstract Factory, Factory Method, Singleton, Observer, Objetifier, Template, Proxy e Distributed Proxy.

ABSTRACT

Many tools, techniques and technologies have been developed to deal with problems in the design of object oriented software. It is argued that design patterns can be used as a basic tool to the design of highly reusable software.

A main concern about design patterns is taken into consideration in the work presented in this dissertation:

Are design patterns a good tool to obtain flexible and reusable cooperative systems?

For this purpose, a virtual conversation system on the Internet running on the Java platform of the Sun Microsystems was developed by using a design methodology totally based on design patterns. The level of reusing of the system was evaluated by extending it with new functions and by modifying part of its interaction structure.

The design patterns used in this work are the following: Abstract Factory, Factory Method, Singleton, Observer, Objectifier, Template, Proxy and Distributed Proxy.

SUMÁRIO

LISTA DE FIGURAS

CAPÍTULO 1

INTRODUÇÃO

O principal objetivo do projeto de software orientado a objeto é a reutilização. Inúmeros mecanismos têm sido propostos ao longo das últimas décadas para permitir, facilitar e orientar o projeto de software flexível e reutilizável. Muitos deles, por exemplo, classes, herança, polimorfismo, tipos abstratos de dados, etc., são diretamente disponíveis a nível de linguagem de programação. Infelizmente, a concepção de software flexíveis e reutilizáveis é um dos objetivos de projeto mais difícil de ser alcançado.

Por outro lado, o processo de concepção de um software é geralmente acompanhado por um ganho de experiência a medida que novas soluções são propostas. Muitas dessas soluções são reutilizadas em diferentes tipos de projetos de software formando o que se convencionou chamar de *padrões de design* [GHJV95]. Os padrões de design surgiram uma como ferramenta de auxílio a concepção que relaciona um problema presente em uma variedade de software a uma solução comprovada pela experiência. O principal desafio na utilização de padrões de design é entendê-los e aplicá-los corretamente para conceber software que sejam flexíveis e reutilizáveis.

Entre os muitos fatores que determinam o grau de reutilização de um software encontram-se o nível de acoplamento semântico entre os objetos participantes e o tipo de cooperação que é formada entre eles. Os relacionamentos entre as entidades constituintes de um software indicam o nível de acoplamento semântico do software. Por exemplo, a relação classe/subclasse provoca um alto grau de acoplamento entre os objetos das classes mãe e filha, pois qualquer modificação na classe mãe afeta geralmente os objetos da classe filha. Os padrões de design orientam o projeto para soluções exibindo um mínimo de acoplamento.

Objetos cooperam para o cumprimento de uma atividade. A cooperação pode ser efetuada de várias maneiras. Muitas vezes, a forma básica de cooperação entre objetos presente nas linguagens de programação orientada a objetos, invocação de métodos, não é adequada. É preciso encontrar soluções que permitam um certo grau de reutilização. Por exemplo, um objeto servidor fornece um objeto ligação para as solicitações de serviço de um objeto cliente. A esse respeito, existem padrões de design fornecendo um bom número de mecanismos de cooperação entre objetos. O projetista deve escolher aquele (ou aqueles) que fornecer a melhor solução em termos de reutilização.

Os padrões de design têm sido utilizados ao longo dos últimos anos, em particular, após a publicação do livro "Design Patterns: Elements of Reusable Object-Oriented Software" [GHJV95] no projeto de uma grande variedade de softwares. Entretanto, relatos de experiências bem sucedidas na área de sistemas cooperativos ainda são relativamente raros. O objetivo principal deste trabalho é demonstrar a validade dos padrões de design na busca de soluções flexíveis e reutilizáveis em sistemas cooperativos em o ambiente aberto. Para tanto, abordamos três temas principais: os padrões de design, as sistemas distribuídos e o desenvolvimento de um sistema cooperativo a partir de uma metodologia totalmente centrada em padrões de design. Este último, trata-se de um sistema de conversação virtual sobre o ambiente Internet utilizando a plataforma Java da Sun Microsystems.

Esta dissertação está organizada da seguinte maneira:

- No próximo capítulo, apresenta-se os padrões de design, conceitos, idéias e exemplos envolvendo alguns dos principais padrões de design conhecidos.
- No terceiro capítulo, aborda-se alguns dos principais aspectos dos sistemas distribuídos, em particular, os sistemas baseados em objetos distribuídos com troca de mensagens.
- No quarto capítulo, os padrões de design são utilizados no projeto de um sistema de conversação virtual sobre o ambiente Internet.
- No quinto capítulo, apresenta-se dois aspectos de reutilização do projeto apresentado no quarto capítulo, um funcional, estendendo-o com mais duas funcionalidades, e um estrutural, modificando sua estrutura de interação.
- Por fim, no sexto capítulo, as conclusões e as perspectivas de continuação deste trabalho.

CAPÍTULO 2

PADRÕES DE DESIGN

2.1. Introdução

A tecnologia de concepção design de software vem continuamente se desenvolvendo através do surgimento de novas técnicas e do aperfeiçoamento das já existentes. O projeto orientado a objeto é uma das técnicas de design de software que mais tem avançado ao longo dos últimos anos. Seu maior postulado é que objetos podem ser reutilizados. Este postulado entretanto não tem sido suficientemente convincente quando os sistemas crescem em complexidade e tamanho. Esse e outros problemas têm conduzido muitos pesquisadores a estudar novas técnicas de projeto de software.

Os padrões de design emergiram como uma das técnicas mais promissoras para alcançar uma melhor qualidade de software. A publicação do livro "*Design Patterns: Elements of Reusable Object-Oriented Software*" (Padrões de designs: Elementos de Software Orientado a Objetos Reutilizável) [GHJV95], considerado um marco neste tipo de tecnologia, descreve um catálogo com vinte e três padrões de design.

Um padrão de design é a descrição de uma solução para um problema de design de software em um determinado contexto. O projeto de software OO reutilizável implica na pesquisa dos objetos pertinentes ao problema, conversão das classes a um grau de granularidade correta, definição da hierarquia de classes e o estabelecimento de

relações chaves entre elas. Os padrões de design orientam o projeto de software para a reutilização das soluções adotadas e flexibilizam o projeto para a incorporação de novas funcionalidades

Os padrões de design foram inspirados no trabalho feito pelo arquiteto Christopher Alexander que estudou formas para facilitar o processo de design de construção de edifícios e áreas urbanas. Alexander definiu os padrões de design da seguinte forma:

"Cada padrão de design é uma regra de três que expressa uma relação entre um contexto, um problema e uma solução".

É daí que vem a definição mais popular de um padrão de design :

"Uma Solução para um Problema num Contexto".

O ciclo para o desenvolvimento de abstrações individuais, proposto por Mary Shaw, nos ajuda a entender melhor onde os padrões de design atuam no projeto de software:

"Primeiro os problemas são resolvidos de maneira *ad hoc*. Com o acúmulo de experiência, algumas soluções funcionam melhor do que outras e uma espécie de folclore é transmitido informalmente de pessoa a pessoa. Eventualmente, soluções úteis são entendidas mais sistematicamente e então codificadas e analisadas. Isto possibilita o desenvolvimento de modelos que suportam a implementação automática e teorias que permitem a extensão e generalização dessas soluções. Surge então um nível mais sofisticado de prática que, por sua vez, permite a abordagem de problemas mais complexos. Estes problemas são então novamente abordados de forma *ad hoc*, reiniciando o ciclo" [MS89].

Este ciclo ocorreu, por exemplo, na definição da abordagem orientada a objetos. Os principais elementos desta abordagem (objeto, classe de objetos e método) já eram utilizados de maneira não sistematizada por diversos projetistas e programadores de software. Estes elementos foram então entendidos mais sistematicamente e a abordagem orientada a objeto foi definida.

Acontece também que os padrões de design são utilizados para obter sistemas mais bem

estruturados. Assim, a utilização de padrões de design que já demostraram sua validade e eficacidade podem garantir em grande parte uma maior produtividade e projetos mais flexíveis e reutilizáveis [GHJV95].

As seguintes datas nos ajudam a visualizar a evolução dos padrões de design na área de software orientado a objeto.

- **1987** - Cunningham e Beck utilizam as idéias de Alexander para desenvolver um pequeno padrão de linguagem para o sistema SmallTalk.
- **1990** - A Gang of Four, GoF, (Gama, Helm, Johnson e Vlissides) iniciam o trabalho de recompilar um catálogo de padrões de design OO.
- **1991** - Bruce Anderson dá o primeiro seminário sobre padrões de design em OOPSLA
- **1993** – Kent Beck e Grady Booch patrocinam a primeira reunião do Hillside Group.
- **1994** - A primeira conferência sobre Padrões em Linguagem de Programação (PLoP)
- **1995** - O grupo GoF publica o livro *Design Patterns: Elements of Reusable Object-Oriented Software*

Um dos padrões de design mais utilizados é o modelo Model-View-Controler de Smalltalk. Este modelo é dividido em três elementos: o *data model que é a* parte do programa, *view* que representa a interface do usuário e o *controller* que interage com o usuário e a *view*. Cada parte representa um problema distinto, cada um com suas próprias regras para manipular os seus dados e cada um se comunica com as outras partes usando apenas um conjunto restringido de conexões. Uma variante deste modelo é utilizado atualmente por Java em seu API JFC Swing.

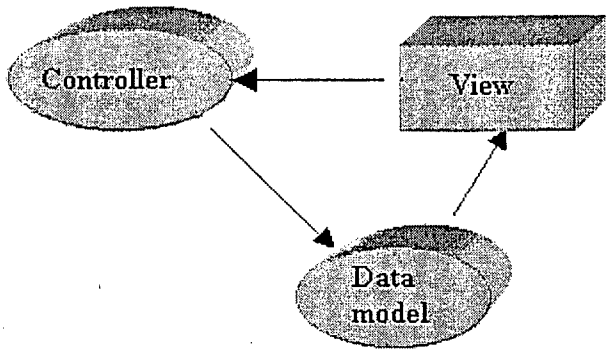


Figura 2.1 Model-View-Controler de Smalltalk

2.2. Tipos de padrões

Existem diferentes tipos de padrões de design para distintas áreas de aplicação: [BRA]

- Análise (apresentado no livro de Martin Fowler).
- Design (apresentado no livro de Erick Gamma).
- Desenvolvimento Organizacional (apresentados nas conferências PloP).
- Processo de Software (apresentados nas conferências PloP).
- Planejamento de projeto (apresentados nas conferências PloP).

Uma outra classificação é dada por Riehle Xullighoven [BRA] em "Understanding and Using Patterns in Software Development" :

- *Padrões Conceptuais,* descritos por meio de termos e conceitos no âmbito da aplicação.
- *Padrões de design,* descritos por meio de elementos de projeto de software tais como objetos, classes, herança e associações.
- *Padrões de programação,* descritos por meio de elementos da linguagem de programação.

Também podem ser classificados por meio de níveis de abstração. Neste caso, podemos citar três níveis, do mais concreto ao mais abstrato:

- Projetos complexos para um sistema ou subsistema
- Solução para um problema de projeto geral num contexto particular.
- Projeto simples de classes reutilizáveis.

2.3. Formas para Padrões de design

Seguem-se as três formas mais utilizadas pela comunidade para descrever padrões de design. A forma Alexander é mais textual e mais apropriada para padrões mais abstratos enquanto que a forma Gamma é mais apropriada para padrões mais próximos do projeto e da realizações de softwares. A forma Coplien é intermediária.

2.3.1. Forma Alexander

A descrição de um padrão é iniciada com a reprodução de **uma fotografia** que descreve um exemplo de uma aplicação do padrão. Em seguida, é descrito um parágrafo inicial que estabelece o **contexto** do padrão. Este parágrafo explica como o padrão ajuda a completar outros padrões mais abrangentes (os padrões "maiores"). Em negrito, é descrito então um parágrafo com a **essência do problema**. O problema é descrito então através de uma discussão com evidências empíricas de sua validade, as várias formas nas quais o padrão pode se manifestar nas construções e outros aspectos. Novamente em negrito é descrito a **solução**. Esta solução é formada pelos relacionamentos físicos e sociais que são necessários para resolver o problema no contexto mencionado. A descrição é sempre na forma de instruções. Esta solução é então ilustrada com um **diagrama da solução**. Finalmente, a descrição é finalizada com a relação dos **padrões "menores"** que são necessários para completar ou embelezar o padrão [CA+77]. Esta forma é utilizada por Alexander em [CA+77]. Variações desta forma são utilizadas, por exemplo, em [RG96].

2.3.2. Forma Coplien

Esta forma foi utilizada pela primeira vez por James Coplien [JC95] constitui-se de seis partes:

- **Problema** , que descreve o problema a ser resolvido.
- **Contexto**, que descreve o contexto no qual a solução descrita resolve o problema.
- **Forças, q**ue apresenta o conjunto de forças que atuam no problema.
- **Solução,** que descreve a solução do problema descrito.
- **Contexto Resultante**, que descreve o contexto resultante após a aplicação da solução.
- **Racionalidade**, que descreve uma racionalidade e exemplos que justificam a solução.

2.3.3. Forma Gamma

Os padrões de design do livro *Design Patterns: Elements of Reusable Object-Oriented Software* [GHJV95] são uma mescla das classificações e níveis de abstrações já mencionados. Neste livro, descreve-se em detalhes um catálogo com vinte e três padrões de design compreendendo os seguintes aspectos:

Nome
Define a sua classificação e descreve sucintamente a essência do padrão de design.

Intenção
Responde perguntas do tipo: o que faz? Qual é o problema resolvido?

Também conhecido como
Quais são os outros eventuais nomes para o mesmo padrão.

Motivação
Descreve um cenário que ilustra o problema e a estrutura de classes e objetos que o resolvem. Normalmente é descrito um exemplo concreto para motivar a sua importância.

Aplicabilidade
Descreve as situações de aplicação.

Estrutura
Descreve uma representação gráfica da estrutura de classes e objetos que o compõe utilizando UML (Unified Modeling Language) [MF97], ilustrando seqüências de pedidos e colaborações entre objetos envolvidos.

Participantes
Descreve as classes e objetos que participam e suas respectivas responsabilidades.

Colaborações
Descreve como os participantes colaboram para a solução do problema.

Conseqüências
Descreve as conseqüências inclusive possíveis limitações da solução.

Implementação
Descreve sugestões para a realização.

Usos conhecidos
Descreve exemplos de utilização.

Código exemplo
Apresenta fragmentos de código para ilustrar como realizá-lo utilizando alguma linguagem de programação.

Padrões relacionados
Descreve quais são os padrões relacionados, suas especificidades e de que forma podem participar no projeto.

2.4. Um catálogo de padrões

Devido a diferenças de granularidade e abstração, foi proposto um catálogo para facilitar a utilização de padrões de design [GHJV95]. Esta classificação é feita de acordo com dois critérios:

Do padrão :

- *De criação*, se o padrão está relacionado ao processo de criação dos objetos.
- *Estrutural,* se o padrão está vinculado a composição de classes ou objetos.
- *Comportamental*, se o padrão define a forma com que os objetos interagem entre si e distribuem as responsabilidades.

Do âmbito:
Refere-se ao fato de que um padrão pode ser aplicado a classes ou a objetos.

A tabela da Figura 2.2 ilustra estas relações.

		Do padrão		
		Criação	*Estrutural*	*Comportamental*
Do Âmbito	Classes	Factory Method	Adapter	Interpreter Template Method
	Objetos	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweigth Proxy	Chain of responsability Command Iterator Mediador Memento Observer State Strategy Visitor

Figura 2.2 Catálogo de padrões de design. [GHJV95]

Segue-se uma breve descrição dos padrões de design citados na tabela da Figura 2.2:

Abstract Factory

Provê uma interface para a criação de famílias de objetos relacionados ou dependentes sem a especificação de suas classes concretas.

Builder

Separa a construção de objetos complexos de suas representações de modo que o mesmo processo de construção possa criar diferentes representações.

Factory Method

Define uma interface para a criação de um objeto, mas transfere para as subclasses a determinação da classe a ser utilizada para a instanciação.

Adapter

Converte a interface de uma classe em outra interface desejada. Permite que classes com interface incompatíveis possam trabalhar juntas.

Bridge

Separa uma abstração de sua implementação de modo que as duas possam ser modificadas independentemente.

Composite

Compõe objetos em uma estrutura de árvore para representar uma hierarquia de objetos. Permite que os clientes tratem objetos individuais e composição de objetos de maneira uniforme.

Chain of Responsability

Evita que o emissor de uma petição se acople a um receptor em particular. No seu lugar encadeia os objetos receptores e passa a petição para a cadeia até que um deles responda a petição.

Command

Encapsula um pedido na forma de um objeto. Permite parametrizar diferentes tipos de pedidos na forma de comandos.

Decorator
Acrescenta responsabilidades adicionais a um objeto dinamicamente.

Flyweight
Utiliza o compartilhamento para suportar o uso de uma grande quantidade de objetos de maneira mais eficiente.

Observer
Define uma dependência um-para-muitos entre objetos. Quando um dos objetos modifica o seu estado, todos os seus dependentes são notificados e atualizados automaticamente.

State
Permite que um objeto altere seu comportamento quando seu estado interno é modificado. O objeto então parecerá que mudou de classe.

Strategy
Define uma família de algoritmos, encapsula cada um dos algoritmos e os torna intercambiáveis. Permite que o algoritmo se modifique independentemente de seus clientes.

Template Method
Define a estrutura de um algoritmo para uma operação transferindo alguns passos desta operação para as subclasses. Permite que uma subclasse redefina certos passos de um algoritmo sem alterar a estrutura do mesmo.

Facade (fachada)
Define uma interface unificada para a representação de um conjunto de interfaces do subsistema, facilitando o seu uso.

2.5. Padrões de design versus frameworks

Um framework fornece uma relação estreita entre os padrões de design e a programação orientada a objeto:

> Um framework é uma mini-arquitetura que provê uma estrutura genérica, funcional e abstrata para uma família de softwares, em geral, com um contexto especificando a colaboração e o uso dos componentes para um domínio específico. [BRA]

Os frameworks não são aplicações, pois carecem de funcionalidade. Entretanto, proporcionam uma infra-estrutura com os elementos (os padrões de design) necessários para que, adicionando-se as funcionalidades próprias da aplicação, pode-se construir aplicações no âmbito do seu domínio.

Assim, um framework é um conjunto de classes cooperantes que compõem um projeto reutilizável para uma classe específica de software. Por exemplo, pode-se gerar um framework para editores gráficos, composição musical ou projeto mecânico. Os frameworks estabelecem a arquitetura da aplicação. Um framework predefine os parâmetros do projeto de modo que o realizador e o projetista se concentrem nos parâmetros específicos da aplicação. Os framework capturam decisões de projeto que são comuns ao domínio da aplicação. Dessa forma, os framework enfatizam a reutilização do projeto através da reutilização do código.

Algumas diferenças entre padrões de design e frameworks são apresentados a seguir:

Frameworks	**Padrões de design**
Incluem código.	São mais abstrato e os códigos são apenas exemplos.
Contêm vários padrões de design.	São pequenas arquiteturas.
Relacionam-se com um domínio tipo de aplicação.	Podem ser utilizados em qualquer de aplicação.

2.6. Descrição de alguns padrões de design

Apresentamos a seguir detalhes de alguns dos padrões de design mais utilizados.

2.6.1. Factory Method

Utilizado freqüentemente em projetos de sistemas orientados a objeto, Factory Method retorna uma classe entre as várias classes possíveis de acordo com o argumento fornecido.

Também Conhecido como

Construtor virtual

Motivação

Os frameworks utilizam classes abstratas para definir e manter as relações entre os objetos. Um framework freqüentemente é responsável também pela criação desses objetos.

Aplicabilidade

Aplica-se o padrão Factory Method nos seguintes casos:

- Quando uma classe não pode prever a classe dos objetos que deve criar.
- Quando uma classe utiliza as suas subclasses para especificar quais objetos serão criados.

Estrutura

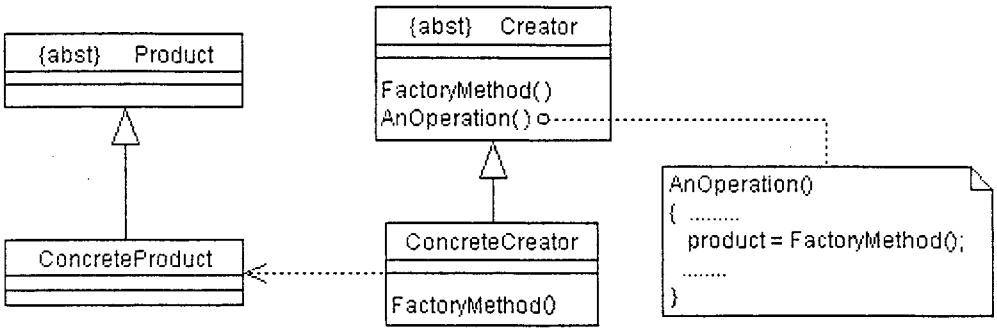


Figura 2.3 Estrutura do padrão Factory Method

Participantes

Product : define a interface de objetos que cria o *FactoryMethod()*.

ConcretProduct : realiza a interface *Product.*

Creator : declara o *FactoryMethod()*, o qual retorna um objeto do tipo *Product.* O *Creator* também pode definir uma implementação por default do *FactoryMethod()* que retorne um objeto por default.

ConcreteCreator: sobrecarrega o *FactoryMethod()* para retornar uma instância de um *ConcreteProduct.*

Exemplo

Utiliza-se o padrão Factory Method no projeto do jogo de labirinto [GHJV95]. Considera-se um labirinto chamado Maze composto de quartos e os quartos compostos de paredes e portas:

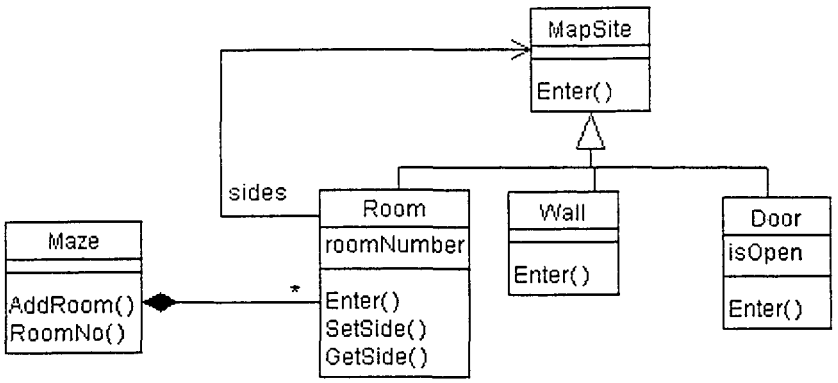


Figura 2.4 Diagrama de classes participantes

Um método para criar um objeto Maze poderia ter a seguinte forma:

```
public class MazeGame
{  ....
   public Maze createMaze()
       {Maze maze = new Maze();
       Room r1 = new Room(1);
       Room r2 = new Room(2);
       Door door = new Door(r1, r2);
       maze.addRoom(r1);
       maze.addRoom(r2);
       r1.setSide(MazeGame.North, new Wall());
       r1.setSide(MazeGame.East, door);
       r1.setSide(MazeGame.South, new Wall());
       r1.setSide(MazeGame.West, new Wall());
       r2.setSide(MazeGame.North, new Wall());
       r2.setSide(MazeGame.East, new Wall());
       r2.setSide(MazeGame.South, new Wall());
       r2.setSide(MazeGame.West, door);
       return maze;
       }
}
```

Esta forma de criar o labirinto Maze não é flexível nem extensível (a razão ficará clara mais adiante). Cria-se então um labirinto mais sofisticado aplicando-se o padrão de design Factory Method aos seguintes métodos: *makeMaze()*, *makeRoom()*, *makeWall()* e *makeDoor()*. O resultado é o seguinte:

```
public class MazeGame
{   .....
    public Maze makeMaze() { return new Maze(); }
public Room makeRoom(int n) { return new Room(n); }
```

```
public Wall makeWall() { return new Wall(); }
public Door makeDoor(Room r1, Room r2) { return new Door(r1, r2); }
public Maze createMaze()
{          Maze maze = makeMaze();
    Room r1 = makeRoom(1);
    Room r2 = makeRoom(2);
    Door door = makeDoor(r1, r2);
    maze.addRoom(r1);
    maze.addRoom(r2);
    r1.setSide(MazeGame.North, makeWall());
    r1.setSide(MazeGame.East, door);
    r1.setSide(MazeGame.South, makeWall());
    r1.setSide(MazeGame.West, makeWall());
    r2.setSide(MazeGame.North, makeWall());
    r2.setSide(MazeGame.East, makeWall());
    r2.setSide(MazeGame.South, makeWall());
    r2.setSide(MazeGame.West, door);
    return maze;
}
}
```

O método *createMaze*() ficou mais complexo, mas agora é mais fácil estender o método para criar um jogo mais sofisticado o *EnchantedMazeGame*:

```
public class EnchantedMazeGame extends MazeGame
{ public Room makeRoom(int n)
  {return new EnchantedRoom(n);
  }
  public Wall makeWall()
  {return new EnchantedWall();
  }
  public Door makeDoor(Room r1, Room r2)
  {return new EnchantedDoor(r1, r2);
  }
}
```

Neste exemplo, as correlações são as seguintes:

- Creator → MazeGame
- ConcreteCreator → EnchantedMazeGame (MazeGame também é um ConcreteCreator)
- Product → MapSite
- ConcreteProduct → Wall, Room, Door, EnchantedWall, EnchantedRoom, EnchantedDoor.

Benefícios

O código é mais flexível e reutilizável devido a eliminação da instanciação das classes

da aplicação. O tratamento é feito utilizando a interface da classe produto.

Colaborações

O *Creator* responsabiliza suas subclasses para a realização do *FactoryMethod*() que, por sua vez, retorna uma instância apropriada de um objeto *ConcreteProduct*.

2.6.2. Abstract Factory

Provê uma interface para a criação de uma família de objetos relacionados ou dependentes sem a especificação de suas classes concretas.

O padrão Abstract Factory é muito similar ao padrão Factory Method. A principal diferença é que o padrão Abstract Factory delega a responsabilidade da instanciação de objetos para outro objeto via composição enquanto o padrão Factory Method utiliza a herança e confia nas subclasses para a instanciação do objeto desejado. Freqüentemente o objeto delegado do padrão Abstract Factory utiliza o Factory Method para realizar a instanciação. Assim, pode-se afirmar que um Abstract Factory retorna um Factory Method entre os vários possíveis.

Também conhecido como

Kit.

Motivação

Uma aplicação clássica de Abstract Factory é o caso quando se deseja suportar múltiplas interfaces de usuários, tais como: windows 9x, Motif ou Machintosh, GUI Factory, etc., que retornam os objetos correspondentes para cada tipo de interface: botões, check boxes e janelas, por exemplo.

Estrutura

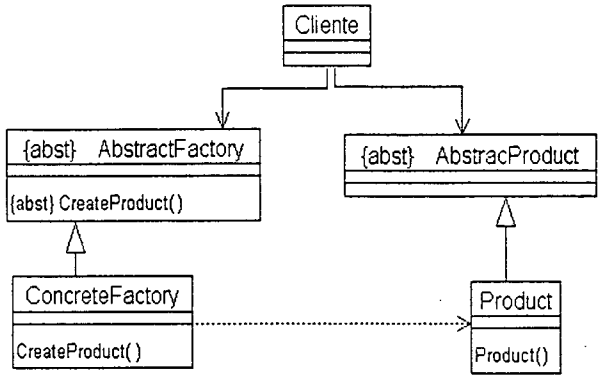


Figura 2.5 Estrutura do padrão de design Abstract Factory

Aplicabilidade

Recomenda-se utilizar o padrão Abstract Factory nas seguintes situações:

- Quando deseja-se conhecer só as interfaces de uma biblioteca de classes e não suas realizações.
- Se uma classe não pode antecipar a classe de objetos que deve criar, por exemplo: ler e editar um arquivo de formato desconhecido.
- Quando o sistema deverá ser independente da forma como os seus produtos são criados, compostos ou representados.
- Quando temos famílias de produtos que são projetados para trabalhar juntos.

Participantes

- *Abstract Factory* : declara uma interface para operações que criam objetos *AbstractProduct.*
- *ConcreteFactory* : implementa as operações que criam objetos *AbstractProduct.*
- *AbstractProduct* : declara a interface para um tipo de objeto *Product.*

- *ConcreteProduct*: define um objeto *Product* a ser criado pelo correspondente *ConcreteFactory* e implementa a interface *AbstractProduct.*
- *Client* : utiliza somente as interfaces declaradas pelas classes *AbstractFactory* e *AbstractProduct.*

Colaborações

- Normalmente uma instância simples da classe *ConcreteFactory* é criada em tempo de execução. Este, por sua vez, cria objetos *Product* possuindo uma realização particular. Para criar diferentes objetos *Product*, o *Client* deverá utilizar diferentes objetos do tipo *ConcretFactory.*
- O AbstractFactory delega a criação de objetos *Product* para a sua subclasse ConcreteFactory.

Exemplo

Aplica-se o Padrão de Design Abstract Factory ao jogo do labirinto Maze [GHJV95]. Primeiro, escreve-se uma classe MazeFactory como segue:

```
/***MazeFactory.
public class MazeFactory
{       public Maze makeMaze()      { return new Maze(); }
        public Room makeRoom(int n) { return new Room(n); }
        public Wall makeWall()      { return new Wall(); }
        public Door makeDoor(Room r1, Room r2) { return new Door(r1, r2); }
}
```

Podemos observar que a classe *MazeFactory* é uma coleção de métodos *Factory Method* e que o *MazeFactory* atua tanto como um padrão *Abstract Factory* como um *ConcreteFactory.* O seguinte código mostra como o método *createMaze* da classe *MazeGame*() toma o *MazeFactory* como um parâmetro:

```
public class MazeGame
{       public Maze createMaze(MazeFactory factory)
        // createMaze delega para o MazeFactory a responsabilidade
        // da criacao dos objetos Maze
        {       Maze maze = factory.makeMaze();
                Room r1 = factory.makeRoom(1);
                Room r2 = factory.makeRoom(2);
                Door door = factory.makeDoor(r1, r2);
```

```
            maze.addRoom(r1);
            maze.addRoom(r2);
            r1.setSide(MazeGame.North, factory.makeWall());
            r1.setSide(MazeGame.East, door);
            r1.setSide(MazeGame.South, factory.makeWall());
            r1.setSide(MazeGame.West, factory.makeWall());
            r2.setSide(MazeGame.North, factory.makeWall());
            r2.setSide(MazeGame.East, factory.makeWall());
            r2.setSide(MazeGame.South, factory.makeWall());
            r2.setSide(MazeGame.West, door);
            return maze;
        }
}
```

Agora pode-se facilmente estender o *MazeFactory* para criar outros objetos *Factory Method.*

```
public class EnchantedMazeFactory extends MazeFactory
{       public Room makeRoom(int n) { return new EnchantedRoom(n); }
        public Wall makeWall() { return new EnchantedWall(); }
        public Door makeDoor(Room r1,Room r2){ return new EnchantedDoor(r1, r2);
}
}
```

Neste exemplo, as correlações são as seguintes:

AbstractFactory	→	MazeFactory.
ConcreteFactory	→	EnchantedMazeFactory (MazeFactory também é um ConcreteFactory).
AbstractProduct	→	MapSite.
ConcreteProduct	→	Wall, Room, Door, EnchantedWall, EnchantedRoom, EnchantedDoor.
Client	→	MazeGame.

Conseqüências

- Desacopla o cliente das realizações das classes concretas.
- Faz o intercâmbio entre a família de produtos de forma fácil, pois o Concretefactory pode suportar uma família completa de produtos.
- Força o uso de produtos pertencentes a uma só família.

Implementações

Um ConcreteFactory pode ter muitas instâncias, mas as aplicações normalmente requerem só uma instância do ConcreteFactory em particular. Nesse caso, é necessário utilizar o padrão Singleton.

Usos Conhecidos

- InterViews GUI Toolkit para gerar objetos look-and-feel específicos.
- ET++ Aplication Framework para obter a portabilidade através de diferentes plataformas.
- Java1.1 AWT e SocketImplFactory.

Padrões Relacionados

- FactoryMethod
- Singleton

2.6.3. Singleton

Restringe a instanciação a um só objeto da classe e provê um ponto de acesso global a esse objeto singular.

Motivação

Em algumas aplicações é necessário a existência de exatamente um objeto de uma classe, por exemplo, a janela de administração da impressora. Essa instância deve ser acessível de maneira simples e fácil.

Aplicabilidade

Utilize o padrão Singleton quando:

- Só deve haver uma instância de uma classe que é acessível aos clientes a partir de um ponto de acesso bem conhecido.
- A instância deve ser extensível por meio de subclasses. Os clientes devem poder utilizar e estender as instâncias sem modificações do código.

Estrutura

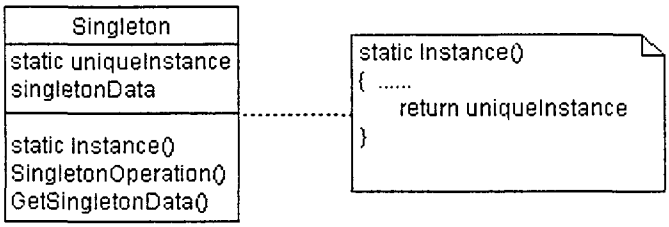


Figura 2.6 a classe do padrão Singleton

Conseqüências

Acesso controlado a uma só instância.
Pode ser facilmente modificado para permitir um número variável de instâncias (por extensão).

Exemplo

Exemplo 1 : Singleton sem subclasses [BT98]:

```
public class Singleton
{       // Referencia privada a somente uma instancia
        private static Singleton uniqueInstance = null;
        // Uma instancia de um atributo.
        private int data = 0;

        public static Singleton instance()
        {       if(uniqueInstance == null)
                uniqueInstance = new Singleton();
                return uniqueInstance;
        }

        //O  construtor também é privado
```

```
        private Singleton() {}

        public int getData(){ return data; }
        public void setData(int data){this.data = data;}
    }
```

Exemplo2 : Singleton com Subclasses

Vamos supor que desejamos ter uma só subclasse da classe MazeFactory: EnchatedMazeFactory ou AgentMazefactory. Neste caso há duas soluções possíveis:

Solução 1 : Um método do MazeFactory determina a classe a ser instanciada:

```
public abstract class MazeFactory
{   private static MazeFactory uniqueInstance = null;
      public static MazeFactory instance(String name)
      {   if(uniqueInstance == null)
                if (name.equals("enchanted"))
       uniqueInstance = new EnchantedMazeFactory ();
      else if (name.equals("agent"))
      uniqueInstance = new AgentMazeFactory ();
      return uniqueInstance;
      }

      public static MazeFactory instance()
      {   return uniqueInstance;
      }
}
```

Código do cliente para criar um objeto Factory pela primeira vez:

```
MazeFactory factory = MazeFactory.instance("enchanted");
```

Código do cliente para acessar um objeto Factory :

```
MazeFactory factory = MazeFactory.instance();
```

Como uma opção, pode-se utilizar o nome da classe para gerar uma só instância:

```
public static MazeFactory instance(String name)
{     if(uniqueInstance == null)
      uniqueInstance  = Class.forName (name).newInstance ();
      return uniqueInstance ;
 }
```

Solução 2 : Cada subclasse provê uma método para instanciação estática:

```
public abstract class MazeFactory
{    protected static MazeFactory uniqueInstance = null;

      public static MazeFactory instance()
      {    return uniqueInstance;
      }
}

public class EnchantedMazeFactory extends MazeFactory
{    public static MazeFactory instance()
      {    if(uniqueInstance == null)
      uniqueInstance = new EnchantedMazeFactory();
      return uniqueInstance;
      }
      private EnchantedMazeFactory ()
      {
      }
}
```

Código do cliente para criar um objeto Factory pela primeira vez:

```
MazeFactory factory = EnchatedMazeFatory.instance();
```

Código do cliente para acessar um objeto Factory:

```
MazeFactory factory = MazeFatory.instance();
```

O construtor das subclasses é privativo. Somente uma instância da subclasse pode ser criada.

2.6.4. Observer

Objetivo

Define uma dependência "um para muitos" entre objetos de modo que quando um dos objetos modificar o seu estado todos os seus dependentes são notificados e atualizados automaticamente.

Também conhecido como

Dependentes, Publicar-Assinar, Model-View.

Motivação

A necessidade de manter a consistência entre objetos relacionados faz com que as classes sejam altamente acopladas.

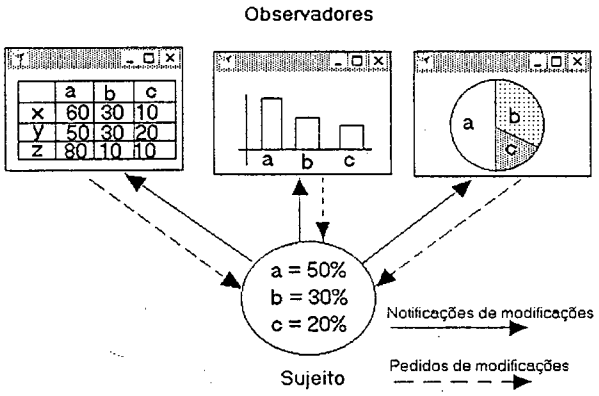


Figura 2.7 Exemplo do padrão Observer.

Aplicabilidade

Utilizar o padrão Observer nas seguintes situações:

- Quando uma abstração têm dois aspectos de modo que um é dependente do outro. O encapsulamento desses aspectos em objetos separados permite-nos modificá-los e reutilizá-los independentemente.
- Quando uma modificação num objeto requer modificações nos outros.
- Quando um objeto deve notificar outros objetos sem a necessidade de conhecê-los.

Estrutura

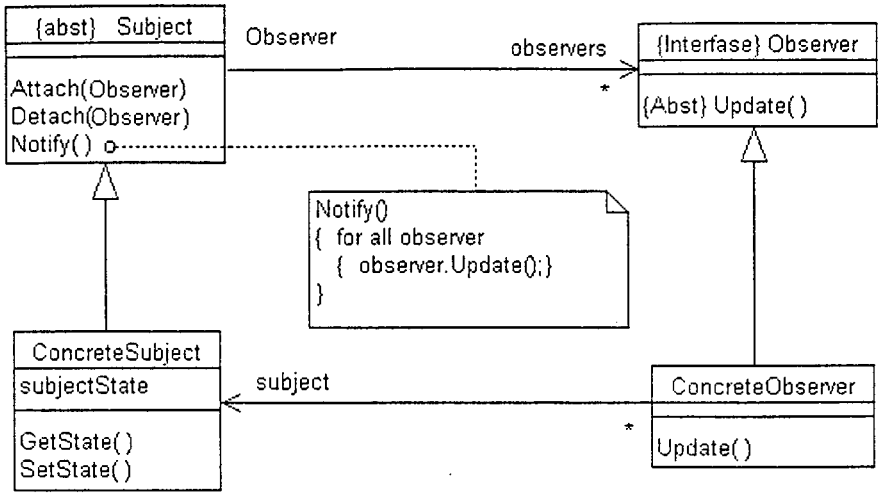


Figura 2.8 Estrutura do padrão Observer .

Participantes

- *Subject* : *conhece os seus observadores, qualquer número de objetos Observer que pode observar um Subject. Provê uma interface para anexar e liberar objetos Observer.*
- *Observer* : *define uma interface de atualização para objetos a serem notificados das modificações do Subject.*
- *ConcreteSubject* : *armazena os estados de importância para os objetos.*
- *ConcreteObserve* : *envia uma notificação para os seus observadores quando o seu estado se modifica.*
- *ConcreteObserver* : *mantém uma referência a um objeto ConcreteSubject. Armazena os estados que devem permanecer consistentes com os dos Subject.*

Colaborações

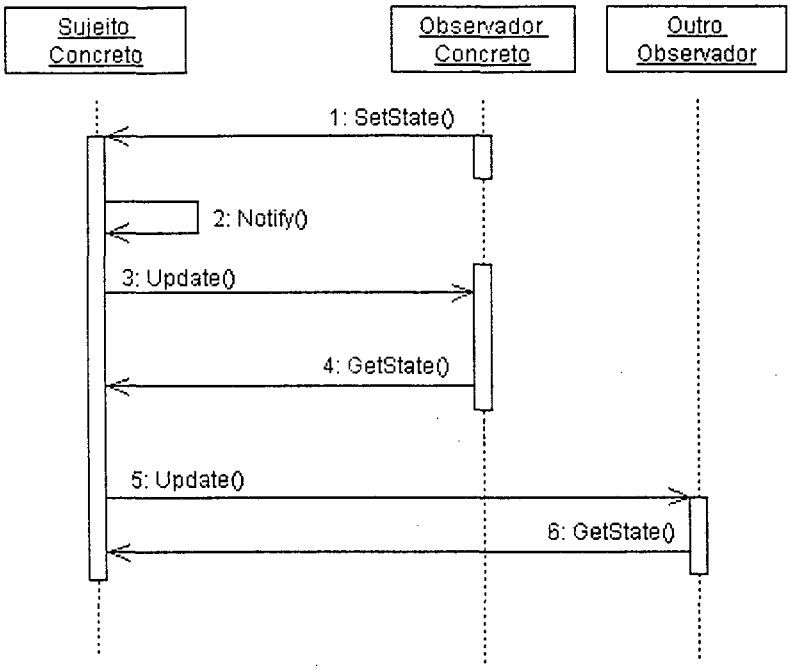


Figura 2.9 Colaboração entre um objeto Subject e dois objetos Observer

Conseqüências

Benefícios:

- Mínimo acoplamento entre o sujeito e o observador.
- Pode-se reutilizar os sujeitos sem reutilizar os seus observadores e vice-versa.
- Pode-se agregar observadores sem modificar o sujeito.
- Um sujeito conhece a sua lista de observadores.
- O sujeito não precisa conhecer a classe concreta de um observador, apenas que cada observador realiza a sua interface de atualização.
- Sujeito e observador podem pertencer a diferentes camadas de abstração.
- Suporte para eventos broadcasting.
- Os sujeitos podem enviar notificações para todos os observadores inscritos.
- Os observadores podem ser agregados/eliminados a qualquer momento.

Responsabilidades:

- Possíveis notificações em cascata. Os observadores não necessariamente estarão atentos sobre o que acontece com os outros observadores, portanto, deve-se tomar cuidado com cada etapa de atualização.

Implementação

Alguns aspectos importantes a considerar na implementação:

- Como o sujeito segue a pista de seus observadores?
- O que acontece se um observador deseja observar mais de um sujeito?
- Quem ativa as atualizações?
- Como tratamos as referências no observador se o sujeito é eliminado?
- Um observador pode inscrever-se a eventos específicos de importância para ele?
- Um Observador pode ser também um sujeito?

Usos conhecidos

O framework de interface de usuário do Model/View/Controler (MVC) de Smalltalk, onde:

- O Model corresponde ao sujeito.
- O View corresponde ao observador.
- O Controler é qualquer objeto que modifica o estado do sujeito.

Outro uso conhecido do padrão observer é o modelo Event de Java e o seu componente "swing" do API JFC .

Padrões Relacionados

Mediador, para encapsular semânticas de atualizações complexas.

Exemplo

Implementação em Java do padrão Observer [BT98].

Poderíamos implementar um observador utilizando as ferramentas básicas da linguagem Java. Entretanto Java provê um suporte para o padrão Observer através da classe *java.util.Observable*. Assim, qualquer classe que deseja ser observada deve estender a classe *java.util.Observable*.. Esta classe prove um conjunto de métodos para agregar/eliminar observadores e um método para notificar os seus observadores de algum evento. Uma subclasse necessita apenas que todos os seus observadores sejam notificados. Outra classe, a classe *java.util.Observer* é a interface das classes do tipo Observador. Por exemplo:

```
// O sujeitoConcreto é a classe a ser observada. Por isso ela estende a
// interface java.util.Observable
//

import java.util.*;

public class sujeitoConcreto extends Observable
{      private String nome;
       private float preço;
       public ConcreteSubject (String nome, float preço)
       {      this.nome = nome;
              this.preço = preço;
              System.out.println ("SujeitoConcreto cria: " + nome + " a "
       +preço);
       }
       public String getNome ()
       {       return nome;
       }
       public float getPreço ()
       {      return preço;
       }
       public void setNome (String nome)
       {      this.nome = nome;
              setChanged ();
```

```
            notifyObservers (nome);
        }
        public void setPreço (float preço)
        {       this.preço = preço;
                setChanged ();
                notifyObservers (new Float(preço));
        }
}


//Teste para sujeitoConcreto, observadorDeNome e observadorDePreço.

public class TestObservers
{       public static void main(String args [])
        {       // Cria um sujeito e três observadores
                sujeitoConcreto         s = new sujeitoConcreto("Zoltrix",1.29f);
                observadorDeNome observNome = new observadorDeNome();
                observadorDePreço observPreço = new observadorDePreço();

                // Agrega observadores para o sujeito
                s.addObserver(observNome );
                s.addObserver(observPreço);
                // Faz  modificações do sujeito
                s.setNome ("Frosted Flakes");
                s.setPreço (4.57f);
                s.setPreço (9.22f);
                s.setNome ("Surge Crispies");
        }
}
```

Saída do teste:

```
SujeitoConcreto cria: Zoltrix a 1.29
O nome atual é : null
O preço atual é :  0.0
O novo nome é : Frosted Flakes
O novo preco é :  4.57
O novo preco é : 9.22
O novo nome é : Surge Crispies
```

Observações :

- Método setChanged() da classe java.util.Observable indica que o sujeito teve o seu estado modificado.

- Método notifyObservers(Objeto Obj) da classe java.util.Observable notifica a modificação aos observadores. Obj é um objeto (neste caso, *nome* do tipo String e *preço* do tipo Float) que é passado para os observadores utilizando o método update(). A Figura 2.10 apresenta um código exemplo.

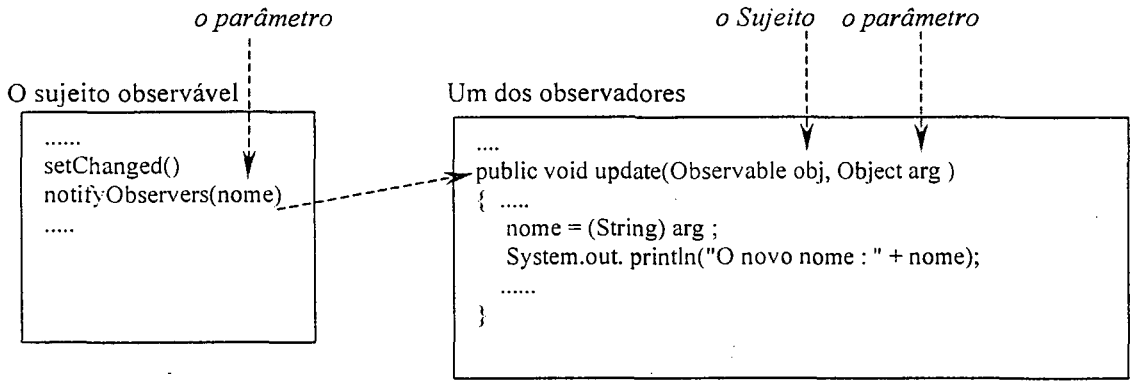


Figura 2.10 Passagem de parâmetros entre o sujeito observado e um dos seus observadores.

```
// Um observador de modificações de nomes

import java. util.*;

public class ObservadorDeNome implements Observer
{       private String nome;
        public ObservadorDeNome ()
        {       nome = null;
                System.out.println ("O nome atual é: " + nome);
        }

        public void update(Observable obj, Object arg )
        {       if (arg instanceof String)   //indaga se o arg corresponde ao
        {               nome = (String) arg ; // tipo nome
                System.out. println("O novo nome é " + nome);
                }
        }
}
```

```
//Um observador de modificações de preços

import java. util.*;

public class ObservadorDePreço implements Observer
{       private float preço;
        public ObservadorDePreço ()
        {       preço = 0;
                System.out.println("O preço atual é:  " + preço);
        }
        public void update(Observable obj , Object arg )
        {       if (arg instanceof Float)              // indaga se o arg
        {       preço = ((Float)arg). floatValue(); // corresponde ao tipo preço;
                        System.out. println("O novo preço é:  " + preço);
                }
        }
}
```

Observações:

- Ambas as classes são observadores e implementam o método update() da interface java.util.Observer.
- No método public void update(Observable obj , Object arg) o *obj* corresponde ao sujeito que está sendo observado e *arg* pode ser um argumento simples correspondente a um objeto básico ou pode corresponder a vários objetos contidos em uma lista de objetos.
- As duas classes recebem todas as modificações ocorridas no sujeito, mas a reação de cada uma depende do tipo de argumento.

CAPÍTULO 3

COMPUTAÇÃO DISTRIBUÍDA

3.1. Introdução

A computação distribuída é a utilizaçõa de técnicas que permitem construir soluções distribuindo uma aplicação em agentes computacionais individuais com capacidade para resolver um tarefa específica que, por sua vez, serão distribuídos ao longo da rede de computadores, compartilhando recursos e realizando tarefas cooperativas. Algumas motivações para construir aplicações distribuídas [JF98]:

- A computação pensada em paralelo permite dividir um grande problema em pequenos problemas sem recorrer ao uso de soluções complexas e dispendiosas.
- É difícil transportar um grande conjunto de dados para o lugar da aplicação. É mais fácil disponibilizar servidores remotos que provêm dados para as aplicações locais.
- Agentes processando informação redundante em diferentes redes podem ser utilizados em sistemas tolerantes a falta. Se uma máquina ou um agente falha, a informação é obtida por redundância.

3.2. Aplicação distribuída concebida em camadas

Uma aplicação pode ser construída em várias camadas, cada uma com um ambiente de trabalho bem definido, ofertando serviços para a camada imediatamente superior.

A camada de mais baixo nível é a camada de rede que conecta um grupo de computadores de modo que eles possam se comunicar. Protocolos de comunicação tais como TCP/IP permitem essa comunicação. Os serviços de mais alto nível estão definidos no topo do protocolo da rede tais como os serviços de nomes e os protocolos de segurança. As aplicações rodam no topo das camadas utilizando os serviços e protocolos de rede e o sistema operacional para coordenar suas tarefas através da rede.

No nível de aplicação, uma aplicação distribuída pode ser constituída das seguintes partes:

Processos

Normalmente um sistema operacional de um computador pode executar vários processos de forma concorrente. Um processo é especificado por um programa em alguma linguagem de programação, compilado e executado utilizando recursos tais como CPU e dispositivos de E/S.

Threads (fluxos de controle)

Todo processo tem pelo menos um thread de controle. Alguns sistemas permitem que um processo tenha vários threads. Cada thread pode ser executado independentemente dos outros threads do programa com ou sem sincronização entre si. Por exemplo, um thread poderia monitorar a entrada de uma conexão socket e outro poderia estar de prontidão para algum evento provocado por um usuário (por meio do teclado ou do mouse).

Objetos

Um objeto é um grupo de dados relacionados com métodos para consultá-los, alterá-los ou para tomar alguma ação neles baseada. Um processo pode ser composto de um ou mais objetos que podem ser acessados por um ou mais threads. Através da introdução da tecnologia de objetos distribuídos tais como RMI e CORBA, um objeto pode ser propagado de forma lógica através de vários processos ou de vários computadores.

Agentes

Um agente é uma entidade computacional mais inteligente e mais autônoma que um objeto. É capaz de cumprir metas de acordo com suas necessidades tal como recuperar informação de grandes bancos de dados. Alguns agentes podem monitorar seus próprios progressos em relação ao cumprimento de suas metas. A definição de agente que utilizaremos neste capítulo é mais informal e mais geral. Utilizaremos o termo agente para referir-nos a um elemento funcional de importância dentro de uma aplicação distribuída. É um componente de alto nível do sistema, definida em torno de uma função especial, utilidade ou papel. Assim, uma aplicação distribuída pode ser composta de um grupo de agentes cooperando para o cumprimento de alguma meta em particular. Cada um desses agentes podem estar distribuídos em vários processos ou hospedeiros remotos e podem consistir de vários objetos e threads. Uma agente também pode servir a mais de uma aplicação ao mesmo tempo.

3.3. Requisitos para o desenvolvimento de aplicações distribuídas

Particionando e distribuindo dados e funções

Uma aplicação pode ser dividida em módulos aplicando-se algum critério de funcionalidade (por exemplo, aplicações de cálculo intensivo) ou pela utilização de dados (por exemplo, aplicações de bancos de dados). Esses módulos devem estar

distribuídos através de máquinas virtuais (compostas por computadores e enlaces de comunicação). Tais módulos precisam ser conectados de maneira fácil, transparente e flexível para permitir o fluxo de dados requerido pela aplicação.

Protocolos de comunicação flexíveis e extensíveis

O tipo e o formato da informação transitando entre os agentes de um sistema distribuído está sujeito a uma alta variabilidade de requisitos. É necessário considerar protocolos de comunicação entre agentes que sejam flexíveis e adaptáveis às mudanças. Por outro lado, quando um agente encontra-se em outro sistema remoto, é necessário que os protocolos de comunicação sejam extensíveis e funcionem adequadamente.

Necessidade de multithread

A disponibilidade para criar vários threads de controle é especialmente importante no desenvolvimento de aplicações distribuídas, pois os agentes são geralmente assíncronos.

Aspectos de segurança

A informação transitando entre agentes deve ser oculta para observação externa. Particularmente, deve-se definir uma forma de autenticação da identidade de um agente, definir seu nível de acesso aos recursos e criptografar os dados.

3.4. Modelos de sistemas distribuídos

3.4.1. Sistema com Memória Distribuída

Sistemas com *Memória Distribuída* são sistemas onde cada processador tem sua própria memória local que pode ser acessada diretamente só pela própria CPU. A transferência de dados de um processador para outro é realizada via rede. Difere dos

sistemas de memória compartilhada nos quais vários processadores têm acesso ao mesmo espaço de memória via um bus de memória.

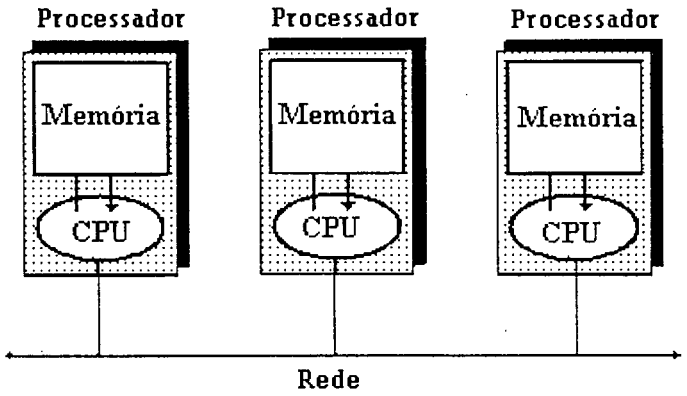


Figura 3.1 Sistemas de Memória Distribuída

O mecanismo pelo qual os dados da memória de um processador é copiado para a memória de outro chama-se *passagem de mensagem.* Nos sistemas de memória distribuída, os dados são geralmente enviados pela rede como pacotes de informação de um processador para o outro. Uma mensagem pode consistir de um ou mais pacotes e, habitualmente, inclui a informação de roteamento e controle.

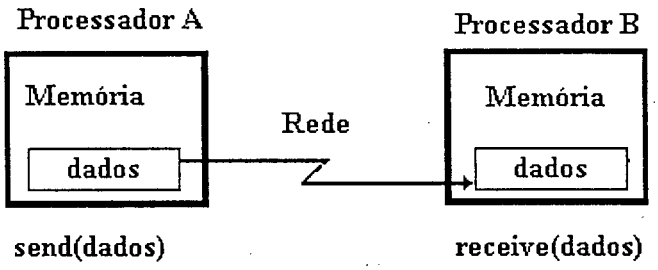


Figura 3.2 Sistemas com passagem de mensagem

Nos sistemas de passagem de mensagem, todos os processos comunicam-se entre si através da troca de mensagem. O envio e a recepção de mensagens são operações cooperantes, *síncronas*, para obter uma comunicação coordenada e segura, ou *assíncronas,* para melhorar o desempenho. Para a comunicação assíncrona, é necessário utilizar um tampão para armazenar as mensagens que são recebidas para serem posteriormente copiadas pela aplicação.

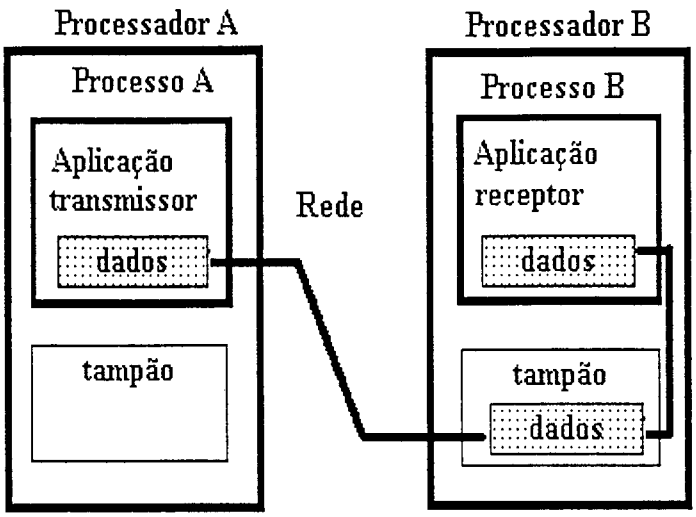


Figura 3.3 Utilização de um tampão para receber mensagens

O MPI (Message Passing Interface) é uma interface projetada para ser um padrão prático e flexível para construção de sistemas de memória distribuída com passagem de mensagem.

3.4.2. Sistemas de Objetos Distribuídos

Os sistemas de objetos distribuídos consistem de um conjunto de objetos remotos, disponibilizados através de rede para que possam ser acessados pelas aplicações distribuídas de forma remota e transparente [RO98]. Os objetos remotos são tratados como se fossem objetos locais. Outra característica é a possibilidade de construir um objeto em um hospedeiro e enviá-lo para outro.

A Figura abaixo ilustra algumas das caraterísticas principais dos sistemas de objetos distribuídos. A especificação da interface de uma classe de objeto que permite gerar uma implementação do servidor do objeto é chamada *skeleton* e uma interface cliente é chamada *stub*. O *skeleton* é usado pelo servidor para gerar novas instâncias da classe de objetos e rotear chamadas remotas aos métodos desses objetos. O *stub* é usado pelo cliente para rotear as transações (os quais são invocação de métodos remotos) para o objeto localizado no servidor. No servidor, as implementações das classes são passadas para um *serviço de registro*, que registra a nova classe no *administrador de objetos* utilizando o *serviço de nomes*. O objeto que é registrado e armazenado no servidor é

disponibilizado para os agentes clientes atraves do *serviço de nomes* e do *administrador de objetos*.

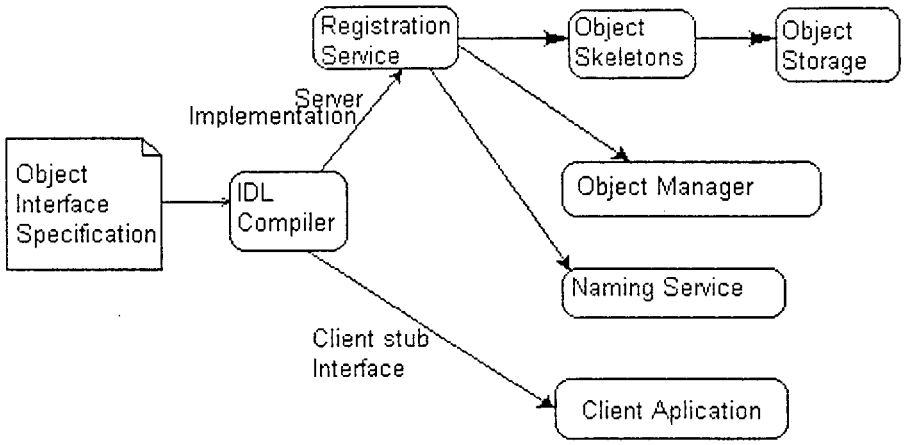


Figura 3.4 Componentes de um sistema de objetos distribuídos [JF98]

A interface de especificação do objeto especifica quais são os métodos do objeto que são acessíveis remotamente e permite aos clientes acessá-los sem conhecer sua implementação e ao servidor implementar a dita classe em alguma linguagem orientada a objetos. Java possui a declaração de interface própria, CORBA utiliza o IDL (Interface Definition Language, uma linguagem independente de plataforma) e DECOM da Microsoft utiliza o COM (Component Object Model).

O administrador de objetos gerencia os *skeletons* dos objetos e suas referências no servidor. Este papel é realizado pelo ORB (Object Request Broker) no sistema CORBA e pelo serviço de registro (RMI registry) no sistema Java.

O serviço de registro e *o serviço de nomes* é o intermediário entre o objeto cliente e o administrador do objeto. Uma interface para um objeto é registrado para que ele seja acessível ao cliente. Este, por sua vez, utiliza o serviço de nomes para criar e utilizar remotamente os objetos que precisa.

O *protocolo de comunicação de objeto* gerencia os pedidos remotos e deve suportar pelo menos o envio e recepção de referências aos objetos, referências aos métodos e aos dados na forma de objeto ou tipo básico.

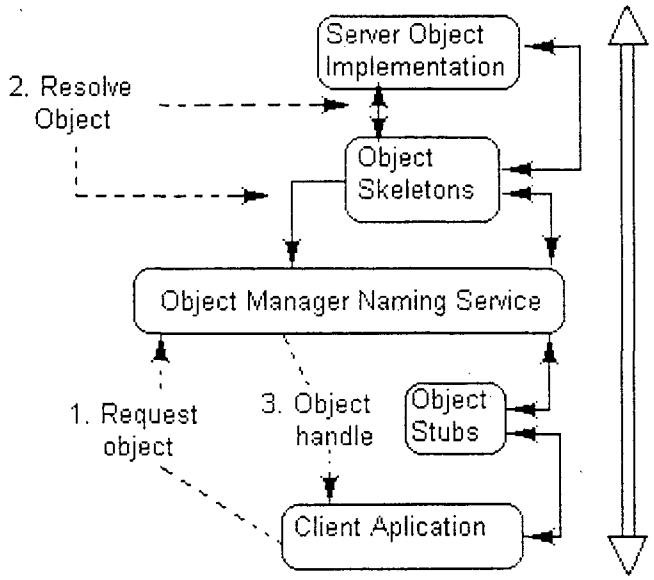


Figura 3.5 Transações em tempo de execução [JF98]

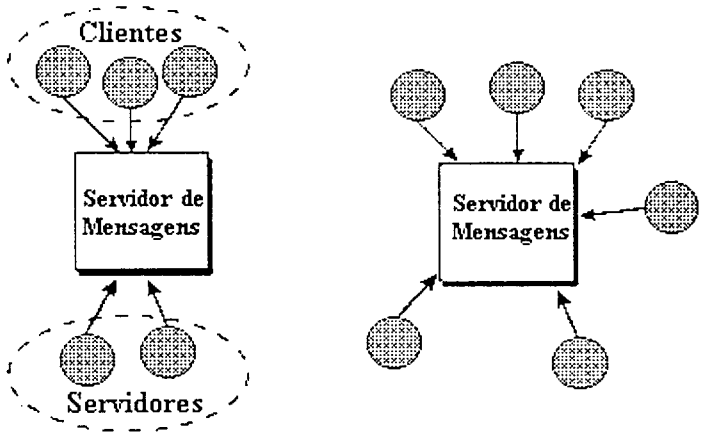
A Figura 3.5 mostra as transações em tempo de execução. O cliente solicita ao administrador de objetos e ao serviço de nomes, uma nova instância de uma classe de objeto. Por conseqüência, um handler do objeto gerado é entregue para a aplicação cliente. Com este handler (uma referência ao objeto) o cliente pode interagir com o objeto.

3.4.3. Aplicações Distribuídas com passagem de mensagens.

Geralmente as aplicações distribuídas com passagem de mensagem são implementadas utilizando um servidor de mensagens para que os componentes da aplicaçõa possam comunicar-se entre si. Todas as mensagens são enviadas ao servidor que, por sua vez, os envia aos componentes destinatários atuando de maneira similar a um serviço postal. As mensagens são eventos, petições e respostas que são criadas e enviadas por um componente da aplicação distribuída para outro. O servidor de mensagens podem ser configurado em um dos seguintes esquemas:

- Cliente/servidor e
- Ponto a ponto.

A plataforma Java possui um API para implementar aplicações distribuídas com passagem de mensagens: o JMS (Java Messaging Framework). Ele provê uma API simples e unificada suportando mensagens que contêm objetos e páginas XML(eXtensible Markup Language). O JMS implementa o modelo ponto a ponto e o modelo publicar/assinar (publish/subscribe). Eles representam dois grandes paradígmas de sistemas com passagem de mensagens.



A) Cliente / servidor B) Comunicação ponto_a_ponto

Figura 3.6 Esquemas de sistemas com passagem de mensagens

Vantagens da utilização de mensagens

A utilização de mensagem na construção de aplicações distribuídas permite um baixo nível de acoplamento entre os componentes. Isso faz com que o sistema seja mais modular, mais aberto para a reutilização e mais confiável. Além de isso, provê escalabilidade, pois novos componentes podem ser agregados de maneira simples, tanto pelo lado do cliente como do servidor. O volume de mensagens pode aumentar sem provocar modificações na aplicação.

Por outro lado, a utilização de um servidor de mensagens facilita as tarefas do projeto

do sistema pelas seguintes razões:

- Permite priorizar as mensagens;
- Envia as mensagens de forma síncrona ou assíncrona;
- Garante que as mensagens sejam enviadas só e somente uma vez;
- Suporta notificação do envio de mensagens;
- Suporta tempo de vida das mensagens;
- Suporta transações.

A mensagem

Essencialmente uma mensagem é uma estrutura de dados contendo a informação que é enviada de um agente para outro sobre um canal de comunicação [JF98]. Algumas mensagens são petições, outras contêm dados e notificações para outro agente. Geralmente uma mensagem é composta de um identificador e de argumentos. O identificador especifica o tipo de mensagem e os argumentos entregam a informação adicional que é interpretada baseado no tipo de mensagem. Os identificadores de mensagens são simples, geralmente consistem de uma palavra ou de um inteiro representando um tipo de mensagem em particular. O tipo de mensagem corresponde a um tipo de serviço fornecido pelo agente receptor. Quando um agente receptor recebe uma mensagem ele consulta uma tabela para saber o tipo da mensagem para que possa traduzir os seus argumentos adequadamente. Os argumentos podem ser de vários tipos, dos mais básicos aos mais complexos. A quantidade de argumentos para uma mesma classe de mensagens pode ser variável ou fixa.

Processamento de mensagens

Na construção de um sistema de agentes com troca de mensagens, é importante pensar de que forma o processamento das mensagens se integra aos objetos gerenciados pelo agente. Neste sentido, é desejável considerar as seguintes linhas de projeto:

- Separar os detalhes de comunicação dos detalhes da aplicação. Isto permite desenhar a maioria das classes em relação aos aspectos próprios da aplicação e não com os aspectos de comunicação entre agentes.

- Prover uma maneira estruturada de acesso aos métodos dos objetos da aplicação.
- Permitir que o agente tenha a capacidade de enviar, receber e processar mensagens de maneira assíncrona.

3.4.4. Sistemas cooperativos

Um sistema cooperativo é composto de um conjunto de agentes locais ou remotos compartilhando informação, enviando petições e interagindo entre si para a obtenção de uma meta comum. Um sistema cooperativo possui os seguintes elementos:

- Agentes autônomos ou conduzido por usuários;
- Servidores de operações e serviços;
- Repositório de dados dinâmicos e persistentes;
- Transações entre agentes, servidores e dados.

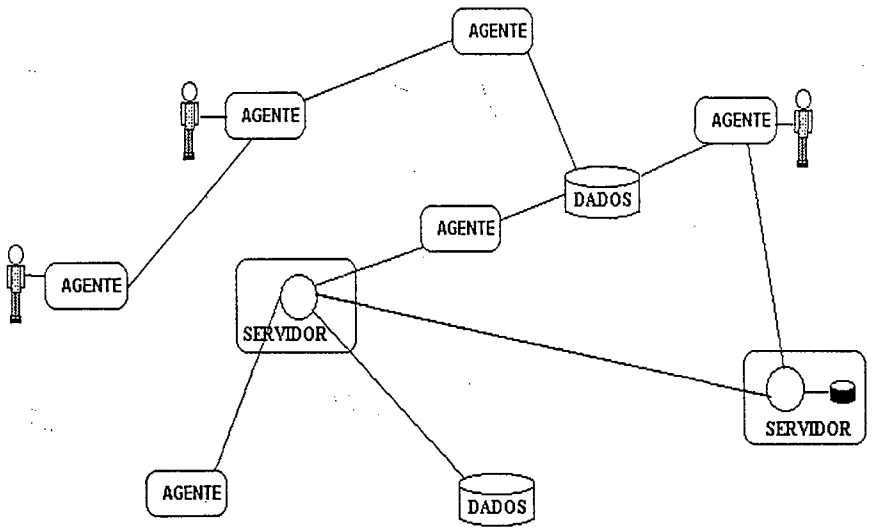


Figura 3.7 Estrutura de um sistema cooperativo

Todos eles ocorrem em qualquer sistema distribuído, mas o fato de realizarem transações para a obtenção uma meta comum faz com que o sistema seja cooperativo. Exemplos de sistemas cooperativos típicos:

- Tela compartilhada;
- Conversação interativa;
- Máquinas de computação paralela e distribuída;
- Agentes de pesquisa coordenada de dados.

Num sistema cooperativo, os agentes interagem dinamicamente. Por isso, a comunicação deve ser flexível e com capacidade de rotear as transações. Dependendo da aplicação, deve suportar mensagens ponto-a-ponto entre agentes, envio de mensagens broadcast para toda a comunidade ou para um grupo específico de agentes participantes. Além de disso, deve contar com alguma forma de identificação para diferenciar um agente do outro e para o correto envio e recepção de mensagens e também com alguns mecanismos de autenticação.

A colaboração entre agentes é expressa normalmente por um conjunto de dados que necessitam ser compartilhado. Um aspecto importante no projeto de tais sistemas é manutenção da integridade e consistência do estado da informação compartilhada. Normalmente um agente atua como um intermediário para tratar os eventos que afetam o estado do sistema. O mediador recebe as notificações por parte do agente e leva cabo as atualizações de estado necessárias. Isto permite uma seqüência correta e segura das modificações dos estados compartilhados.

CAPÍTULO 4

UTILIZANDO PADRÕES DE DESIGN NO PROJETO DE UM SISTEMA DE CONVERSAÇÃO VIRTUAL SOBRE A INTERNET

4.1. Introdução

Os capítulos precedentes, mostraram que os padrões de design de software são utilizados para a concepção de aplicações flexíveis e reutilizáveis. Viu-se alguns aspectos relevantes da arquitetura de sistemas distribuídos baseados em mensagens e em objetos distribuídos. Neste, apresentamos o desenvolvimento de uma aplicação distribuída denominada Sistema de Conversação Virtual sobre a Internet (SCV) utilizando a plataforma Java da Sun Microsystems.

Os padrões de design desempenham o papel principal no desenvolvimento desta aplicação. São utilizados para proporcionar um design flexível e reutilizável tanto nos aspectos funcionais quanto nos estruturais. Vislumbra-se a longo prazo o estabelecimento de uma metodologia centrada em padrões de design para o desenvolvimento de aplicações cooperativas sobre a Internet.

A aplicação desenvolvida neste trabalho não representa por si só o objetivo principal deste trabalho. Aliais existe um certo número de realizações, até comerciais, disponíveis. Foi escolhida por ser típica e representativa de uma classe de aplicações cooperativas sobre a Internet. Por isso, apenas alguns aspectos relevantes para o estudo apresentado nesta dissertação foram considerados no desenvolvimento do SCV.

O desenvolvimento do SCV possui três vertentes principais:

Funcional
Relativo à forma pela qual a aplicação é descomposta em elementos funcionais.

Estrutural
Relativo aos mecanismos de concorrência e cooperação entre as atividades envolvidas na aplicação.

De comunicação
Relativo à estrutura de comunicação suportando as diferentes necessidades de comunicação de dados entre componentes da aplicação.

Com relação a esta última, foi utilizada a estrutura de comunicação baseada no padrão TCP/IP diretamente disponível na plataforma Java. Sendo esse um assunto relativamente banal e fora do foco do estudo desta dissertação, nenhum detalhamento maior é apresentado aqui. O leitor encontrará em [DOU94, SUN] uma apresentação mais detalhada a esse respeito. O resto deste capítulo é dedicado à apresentação das outras duas vertentes.

A noção de agente utilizada neste capítulo é informal e um pouco mais geral que o habitual. Empregamos o termo agente para referir-nos a um elemento funcional de importância dentro de uma aplicação distribuída. É um componente de alto nível do sistema definido em torno de uma função especial. Assim, uma aplicação distribuída pode ser composta por um grupo de agentes colaborando para o cumprimento de uma meta comum.

Associado a noção de agente encontramos a de mensaegens. Mensagens correspondem à informação que é enviada de um agente para outro, aos serviços que um agente

servidor presta aos agentes usuários, às atividades que um agente deve executar, às notificações de eventos, ao controle de ingresso ao sistema e assim por diante. É uma peça de informação necessária para a comunicação entre dois ou mais agentes.

4.2. Estabelecimento dos requisitos da Aplicação

Os requisitos funcionais especificam aspectos próprios da aplicação e estão mais relacionados com o que um usuário espera do sistema. Os requisitos não funcionais especificam aspectos estruturais e são transparentes ao usuário.

Requisitos funcionais

1. O sistema SCV deve ser projetado e construído para funcionar em ambiente distribuído e aberto para que possa ser acessível a partir de qualquer computador ligado à Internet.
2. Um usuário pode ser habilitado para as seguintes funções:
   - Criar uma nova sala.
   - Trocar de sala.
   - Enviar mensagens publicadas na sala.
   - Receber as mensagens da sala a que pertence.
3. A troca de sala deve ser realizada de forma transparente ao usuário.

Requisitos não funcionais

1. O sistema SCV deve ser do tipo distribuído, colaborativo e com passagem de mensagem
2. O sistema deve ser executado no ambiente Internet.
3. Deve-se fazer uso de agentes estáticos nos seguintes casos:
   - Agentes usuários para o atendimento ao usuário (um para cada usuário).
   - Agente chat coordenador (único no sistema).
4. O sistema de comunicação de passagem de mensagem deve ser assíncrono.

4.3. Análise dos requisitos e especificações do sistema

Analisa-se alguns aspectos ligados aos requisitos funcionais e não funcionais com o intuito de capturar as classes participantes e suas responsabilidades, os aspectos dinâmicos produtos das interações entre as classes e construir um protótipo de interface gráfica de referência. Para documentar a análise, utilizaremos técnicas de design de sistemas orientados a objetos baseadas na notação UML (Unified Modeling Language) [MF97].

As tabelas abaixo descrevem as responsabilidades e colaborações das principais classes envolvidas. Para distingüir as atividades de um usuário externo (o humano) do processo usuário chamaremos este último de *agente usuário.* O *agente chat* é um processo autônomo que tem a capacidade de interagir dinamicamente com os agentes usuários. Quando o usuário é aceito pelo sistema é gerada uma sessão, uma objeto que tem referência ao agente e sua sala.

Agente usuário	
Solicita o início/término de uma sessão	Agente chat
Envia uma mensagem	Sala
Solicita troca de sala	
Solicita a criação de uma sala	
Solicita a eliminação de uma sala vazia	
Atualiza os eventos da sala	

Agente chat	
Inicia/termina uma sessão	Agente usuário
Publica uma mensagem na sala correspondente	Sala
Efetua a troca de sala de um usuário	
Cria /elimina uma Sala	
Distribui os eventos que acontecem na sala	

Sala	
Incorpora/elimina um agente usuário na sala	Agente chat Agente usuário

Figura 4.1 Descrição das classes participantes, suas responsabilidades e colaborações.

O diagrama "use case" da Figura 4.2 mostra as atividades consideradas importantes para a descrição das funcionalidades do sistema. O ator agente usuário representa um usuário e sua correspondente interface gráfica. O agente chat é o representante do SCV. As atividades *Iniciar/Terminar Sessão* utilizam a classe *Sessão* e as atividades *Criar/Eliminar* e *Trocar Sala*, a classe *Sala*. Fazem parte da interação entre os *agentes usuários* e o *agente chat*.

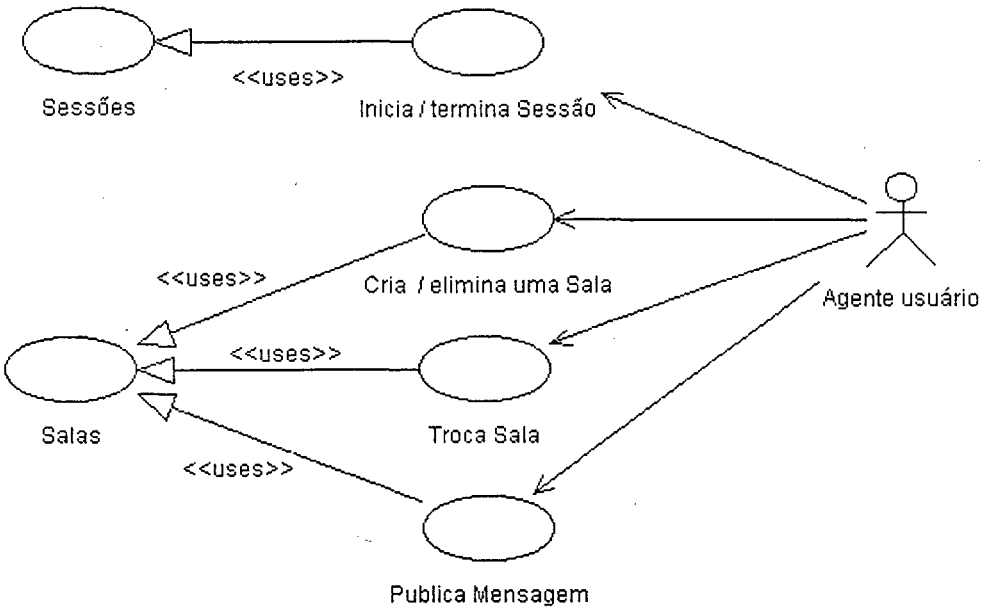


Figura 4.2 Diagrama "use case" do sistema de conversação virtual.

As Figuras 4.3 e 4.4 mostram a seqüência de eventos e atividades que ocorrem na interação entre as classes participantes do SCV. É importante notar que cada vez que acontece um evento os outros participantes são notificados. Isto se deve ao fato de que o SCV é colaborativo: os eventos são enviados por um agente usuário ao agente chat que então distribui aos outros agentes, incluindo o agente emissor do evento.

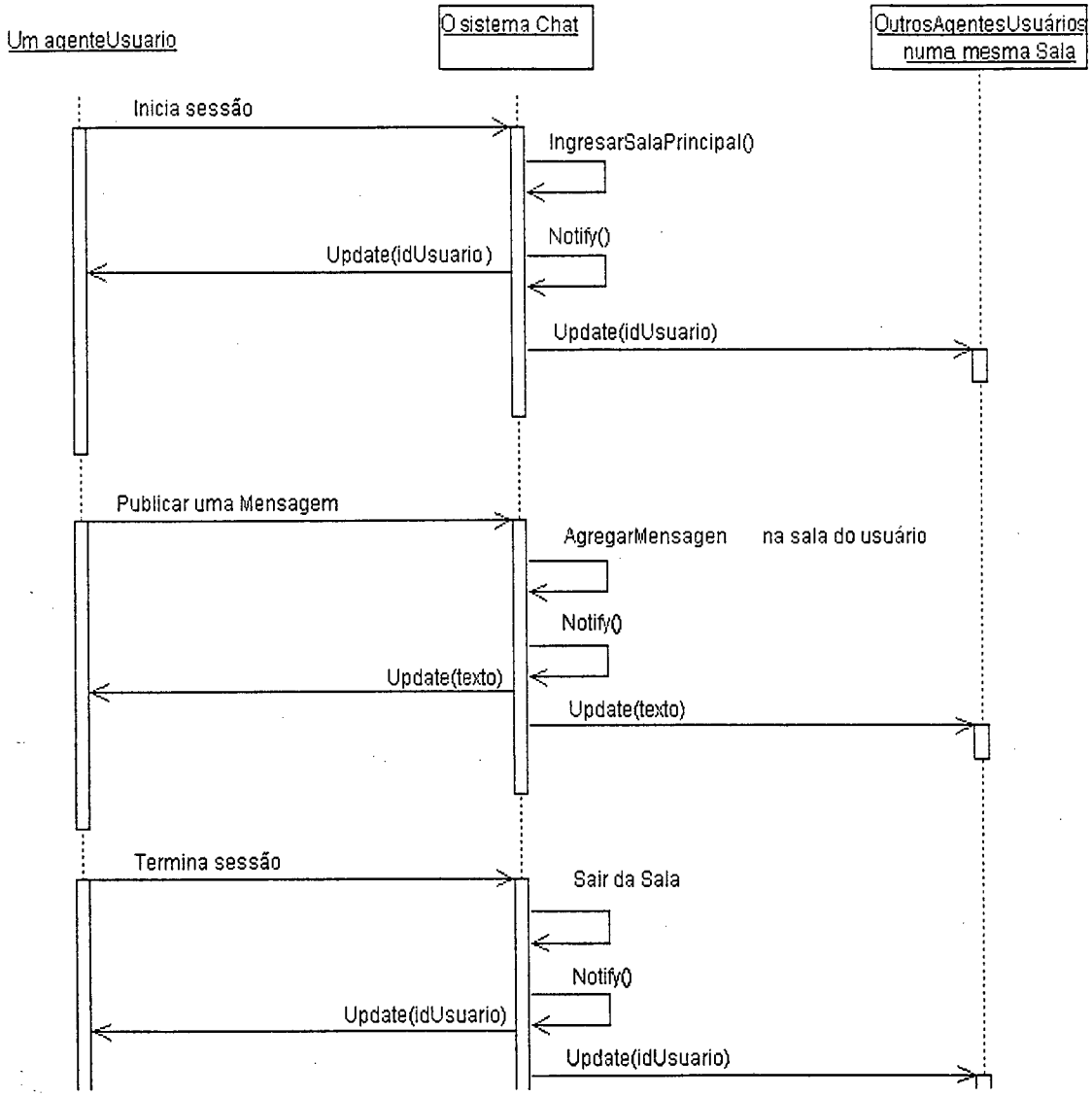


Figura 4.3 Início/término de sessão e envio de mensagens.

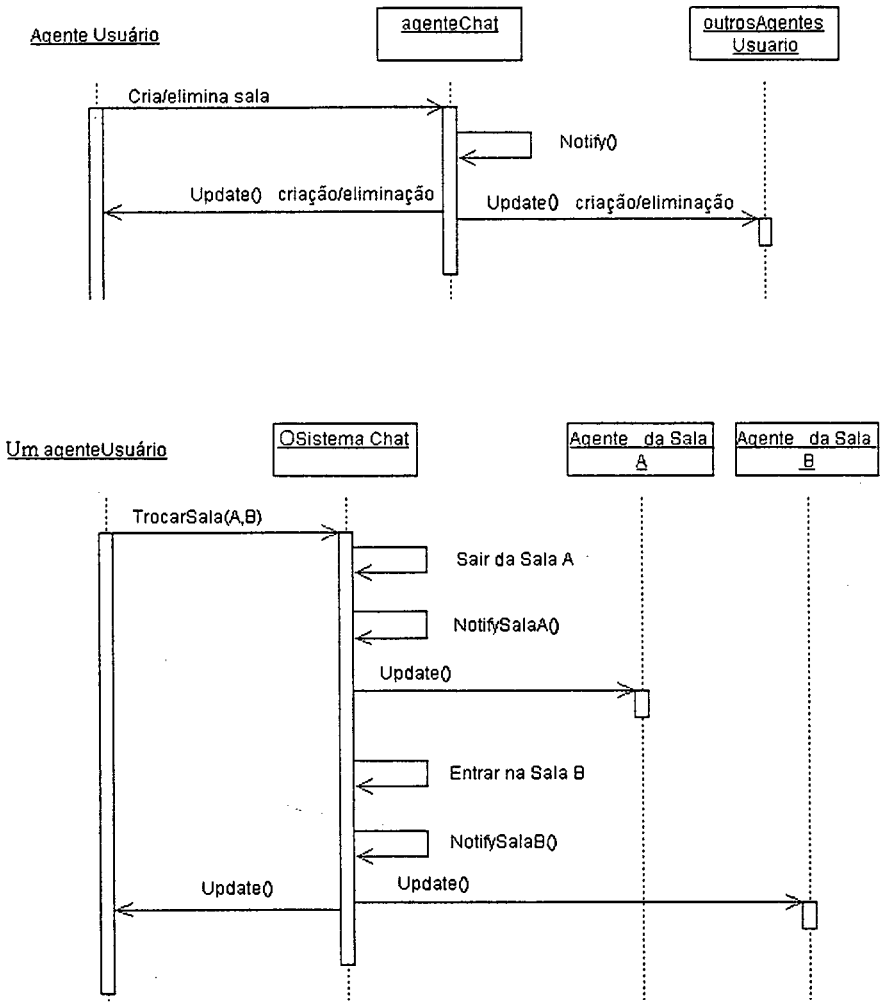


Figura 4.4 Criação , eliminação e troca de sala.

As atividades mostradas na Figura 4.4 correspondem às transações entre o agente usuário e o agente chat. O estado resultante da criação ou eliminação de uma sala é notificado a todos os agentes usuários, mas a troca de uma sala é notificada apenas aos participantes das salas envolvidas. As atividades ligadas a essas atualizações são justamente as que trazem mais complexidade ao design do sistema, pois utilizam os dois tipos de comunicação: síncrona e assíncrona.

4.4. Diagrama de classes do Sistema de Conversação Virtual

A partir da análise dos requisitos podemos esboçar uma primeira tentativa das classes participantes, com seus atributos e métodos. A Figura 4.5 mostra as classes dos agentes, a classe Sessão e a classe Sala. Podemos observar que o agente Chat administra salas e sessões. É ele que registra os eventos ocorrendo no sistema.

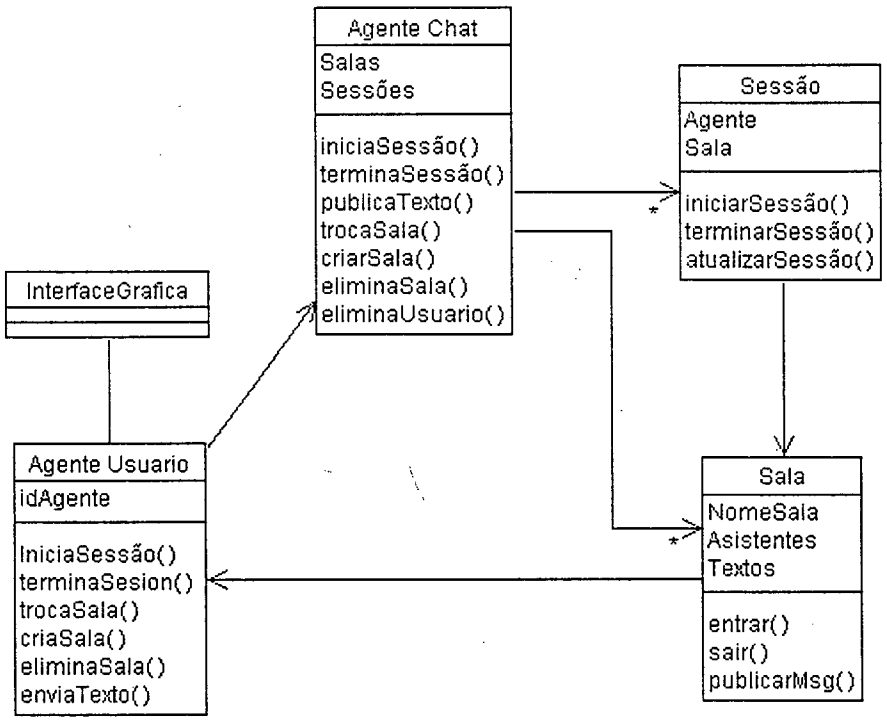


Figura 4.5 Diagrama de classes participantes no SCV.

4.4.1. Protótipo da interface gráfica

A interface gráfica mostrada no Figura 4.6 é composta de botões de comandos que permitem ao usuário trocar, criar e eliminar uma sala além de iniciar e terminar uma sessão.

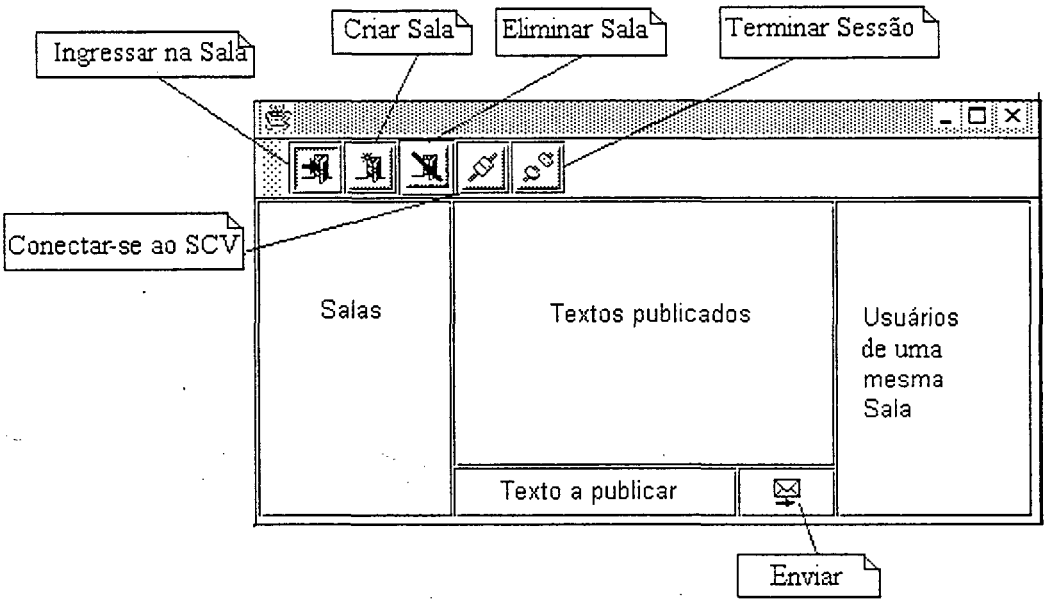


Figura 4.6 Protótipo da interface gráfica do SCV

4.5. Design do SCV

4.5.1. Aspectos de design

A análise dos requisitos apresentada anteriormente não leva em consideração os aspectos relacionados ao design do sistema. Esta etapa aborda os aspectos de software que suportam as atividades do SCV. Em particular, daremos mais atenção aos seguintes aspectos:

- Estrutura da aplicação
- Serviços fornecidos
- Petições remotas de serviços
- Atualizações remotas

Cada um desses aspectos representa um componente do sistema que pode ser tratado de forma insolada e independente [DL98]. O design do sistema completo resulta então da composição dos designs de cada componente.

4.5.2. Estrutura da aplicação

As aplicações distribuídas colaborativas são formadas basicamente por um conjunto de agentes interagindo entre si e compartilhando objetos para atingir uma meta comum. Na aplicação exemplo, temos um conjunto de agentes cuja meta comum é a comunicação interativa em tempo real entre usuários distribuídos ao longo da rede Internet.

Distinguimos duas classes de agentes, o agente chat, com a responsabilidade principal de gerenciar as salas e distribuir a mensagem publicada na sala, e o agente usuário, com a responsabilidade principal de gerar novas mensagens a serem publicadas na sala.

Numa primeira abstração do sistema, podemos visualizar o SCV da forma como é mostrado na Figura 4.7. Não estão sendo considerados os aspectos de comunicação remota nem de passagem de mensagens. Em outras palavras, visualizamos o sistema como se fosse uma aplicação local. Também foi omitido o agente moderador já que esse papel é também desempenhado pelo agente usuário.

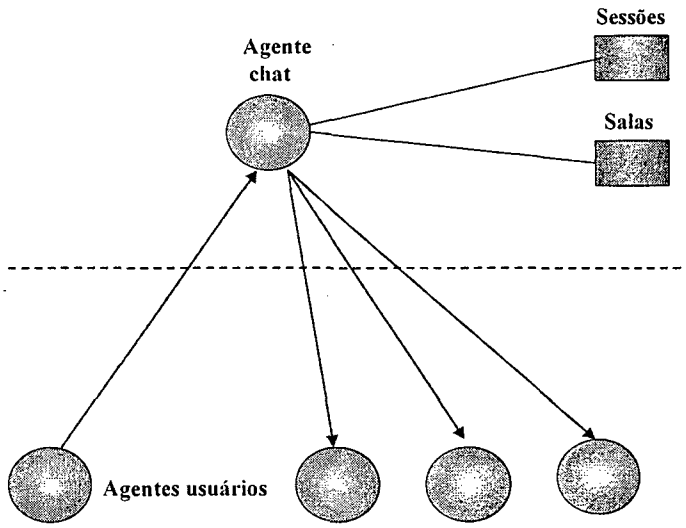


Figura 4.7 Estrutura do SCV.

A estrutura deve contemplar as diferentes atividades do sistema de acordo com os seguintes requisitos funcionais:

- Iniciar/terminar uma Sessão.
- Publicar uma mensagem
- Trocar de sala
- Criar uma sala
- Eliminar uma sala

Esta estrutura será modificada mais adiante quando trataremos os aspectos relacionados a passagem de mensagens e de comunicação remota. O diagrama da Figura 4.8 mostra uma relação simples entre as classes principais.

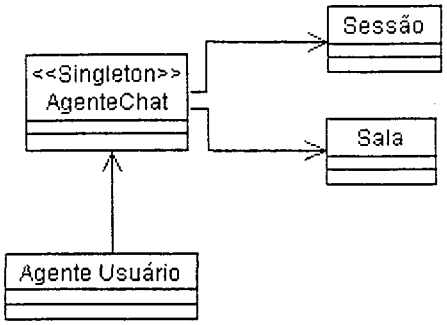


Figura 4.8 Classes participantes na estrutura básica do SCV.

4.5.3. O agente chat e seus serviços

As atividades em questão são na realidade os serviços fornecidos pelo agente chat em relação às salas e às sessões [DL98]. Assim, podemos considerar que os agentes podem ser configurados para realizar um conjunto determinado de atividades de forma seqüêncial ou concorrente podendo envolver zero, um ou mais objetos sujeitos. O principal benefício deste esquema é a possibilidade de modificar uma atividade e incorporar outras sem provocar grandes mudanças no projeto do sistema. Este esquema é mostrado na Figura 4.9.

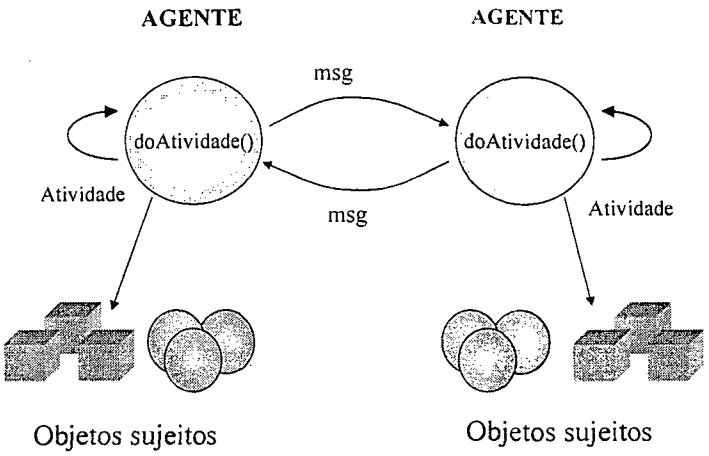


Figura 4.9. Agentes e atividades

A Figura 4.9 mostra dois agentes trocando mensagens entre si. Cada uma é traduzida na forma de atividade a ser efetuada sobre os objetos sujeitos e outros agentes. Cada agente executa atividades através de seu método *doAtividade(msg)*. A ligação com os objetos sujeitos é realizada somente pela execução dessas atividades.

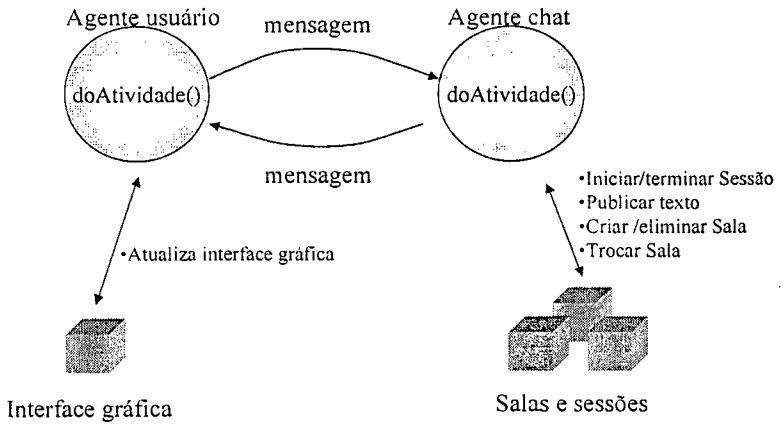


Figura 4.10 Colaboração entre agentes usuários e o agente chat.

A Figura 4.10 mostra um agente usuário e o agente chat com suas respectivas atividades e objetos sujeitos.

Para o design dos agentes usuários e do agente chat utilizou-se os padrões Template de GoF [GHJV95] e Objectifier de W. Zimmer [WZ97]

O padrão Template define um algoritmo base e alguma operação ou tarefa que é delegada a uma subclasse. Sua estrutura básica é mostrada na Figura 4.11.

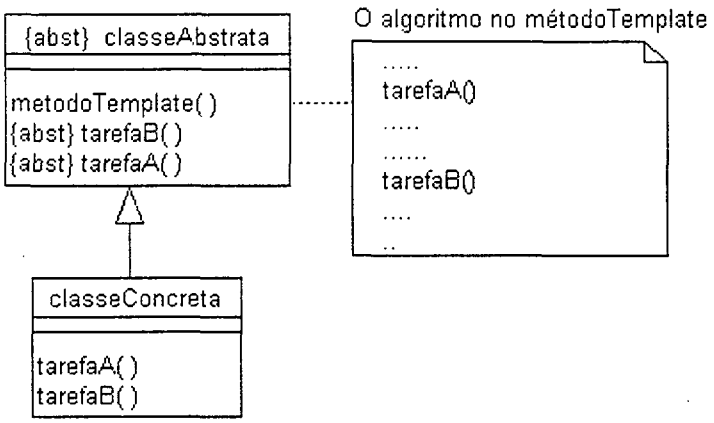


Figura 4.11. O padrão Template [GHJV95]

O padrão Objectifier permite que um objeto modifique o seu comportamento ou propriedades independente dos outros.

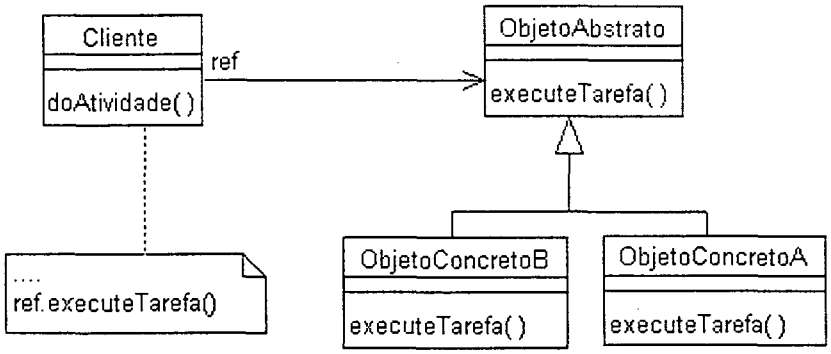


Figura 4.12. Estrutura do padrão Objectifier [WZ97]

O design proposto para o agente chat é uma combinação dos padrões acima. As Figuras 4.13 e 5.14. mostram o design resultante.

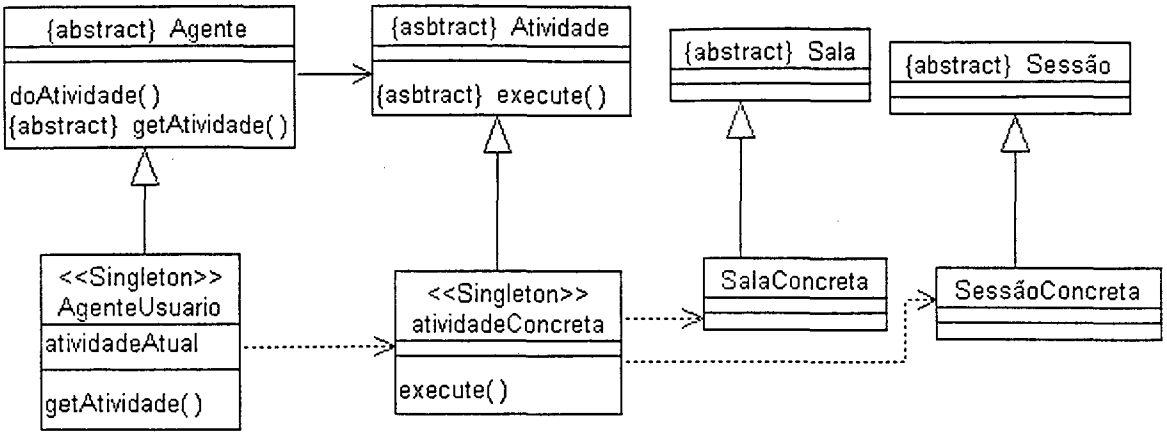


Figura 4.13 A estrutura do agente chat baseado nos padrões Template e Objetifier.

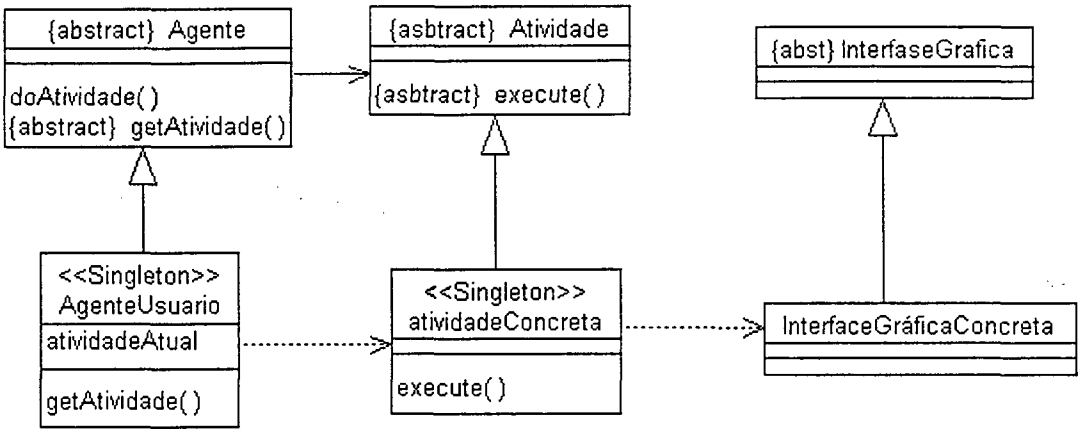


Figura 4.14 A estrutura do agente usuário baseado nos padrões Template e Objectifier.

A realização utilizando a linguagem Java é apresentada abaixo. A classe Mensagem consiste de uma mensagem composta de um código de atividade (obtida pelo método msg.getId()) e os argumentos necessários para executar a atividade. O mensagem é recebida pelo agente que através do método *getAtividade()* obtém a *atividade* a ser executada.

```
public abstract class  Agente
{       private    String idAgente;
        protected Agente agenteRemoto;

        public Agente(String idAgente)
        {   this.idAgente = idAgente;
        }
        public void doAtividade(Mensagem msg)
        {       Atividade atividade = getAtividade(msg.getId());
                atividade.Execute(msg);
        }
        public  abstract Atividade getAtividade(String idAtividade);
        public void doAtividadeRemota(Mensagem msg)
       {       agenteRemoto.doAtividade(msg);
        }
}
```

A classe *Agente* é a classe base para os dois tipos de agentes: o agente chat e o agente usuário. O algoritmo base é o método *doAtividade(Mensagem msg)* e a tarefa abstrata implementada pelo agente concreto é o método *getAtividade(msg.getId())*

```
class  AgenteConcreto extends Agente
{
        private static AgenteConcreto Singleton = null;
        private Receptor receptor;

        public static AgenteConcreto crear(String idAgente)
        {       if(Singleton == null)
                Singleton = new AgenteConcreto(idAgente);
       return Singleton;
        }

        private AgenteConcreto(String idAgente)
        {       super(idAgente);
        }

       public Atividade getAtividade(String idAtividade )
       {       FactoryAtividade factActividade = FactoryAtividade.criar();
               return factActividade.criarAtividade(idAtividade);
       }
}
```

O *AgenteConcreto* gera uma nova atividade através do *FactoryAtividade* seguindo o padrão Abstract Factory [GHJV95]. Uma atividade é uma classe tipo Singleton [GHJV95]. Possui apenas um objeto que só é instanciado quando necessário. O código abaixo mostra a classe abstrata *Atividade* e uma atividade chamada *Atv_PublicaTexto*.

```
public abstract  class Atividade
{           private String  idAtividade;

       public  Atividade(String id)
               {   this.idAtividade = id;
               }
```

```
        public String getId()
        {       return idAtividade;
        }

        public abstract void Execute(Mensagem msg);
}


public class Atv_PublicarTexto extends Atividade
{       private static Atv_PublicarTexto SingleAtividade = null;
        public static Atv_PublicarTexto crear()
        {       if(SingleAtividade == null)
                SingleAtividade = new Atv_PublicarTexto();
                return SingleAtividade;
        }
        private Atv_PublicarTexto()
        {   super("Atv_PublicarTexto");
        }

        public void Execute(Mensagem Msg)
        {       ConjuntoSalas salas = ConjuntoSalas.crear();
                ConjuntoSessoes sessoes = ConjuntoSessoes.crear();

                String idAgente = (String) Msg.getArg(0);
                String texto       = (String) Msg.getArg(1);

                Sessao sessao = sessoes.getSessao(idAgente);
                Sala sala = sessao.getSala();
                sala.addTexto(idAgente+" dice: "+texto);
        }
}
```

Podemos destacar algumas caraterísticas deste design:

- O padrão Singleton é aplicado ao agente chat evitando que existam mais de um agente chat e também para permitir o seu acesso global.
- Não existe um acoplamento entre a classe *AgenteChat* e as classes *Sala* e *Sessão*, pois estão relacionadas somente pelas classes atividades.
- Um novo tipo de serviço pode ser acrescentado livremente mesmo em tempo de execução.
- O agente chat não precisa conhecer a forma como é realizada as classes Sala e Sessão.

4.5.4. Petições remotas de serviços

O *agenteUsuario* é a classe que representa o usuário nas petições de serviços que o *agenteChat* traduz como atividade. Ele poderia solicitar um serviço simplesmente através da invocação do método **doAtividade** () do *agenteChat* da seguinte forma:

agenteChat.**doAtividade** (Mensagem msg);

Isto seria válido se o *agenteChat* e o *agenteUsuario* fossem processos locais. Como lidamos com uma aplicação distribuída, os agentes são executados em computadores geograficamente distribuídos. Por isso, o problema principal do design é conseguir uma abstração funcional para a invocação de um método remoto como se fosse local. Uma solução adotada utiliza o padrão Distributed Proxy [ARS97].

O padrão Distributed Proxy permite desacoplar a comunicação da funcionalidade. A Figura 4.15 mostra um diagrama com três camadas representando a comunicação entre objetos distribuídos. A primeira camada representa interações com invocações normais entre objetos. A Segunda camada introduz os Proxys entre os objetos distribuídos que, por sua vez, convertem referências a objetos em nomes e geram a mensagem correspondente. A terceira camada faz a comunicação física para o envio e recepção em forma de byte. A Figura 4.16 mostra a estrutura do padrão Distributed Proxy.

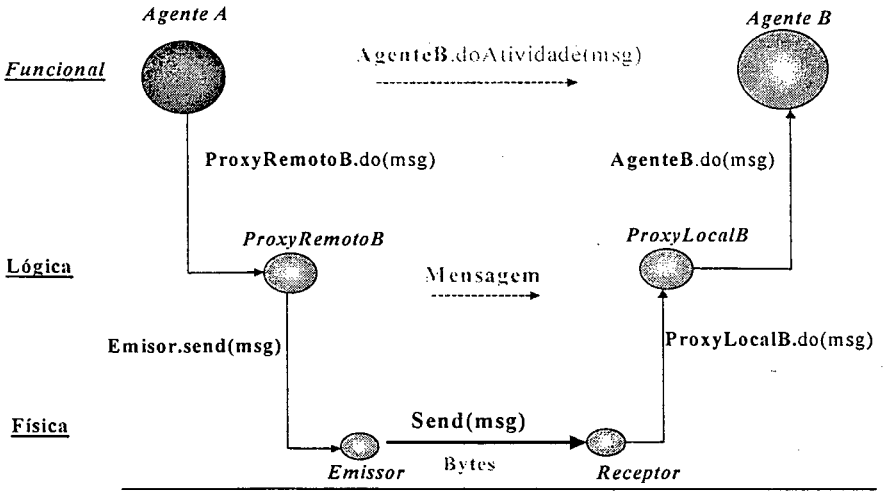


Figura 4.15 O Distributed Proxy em três camadas.

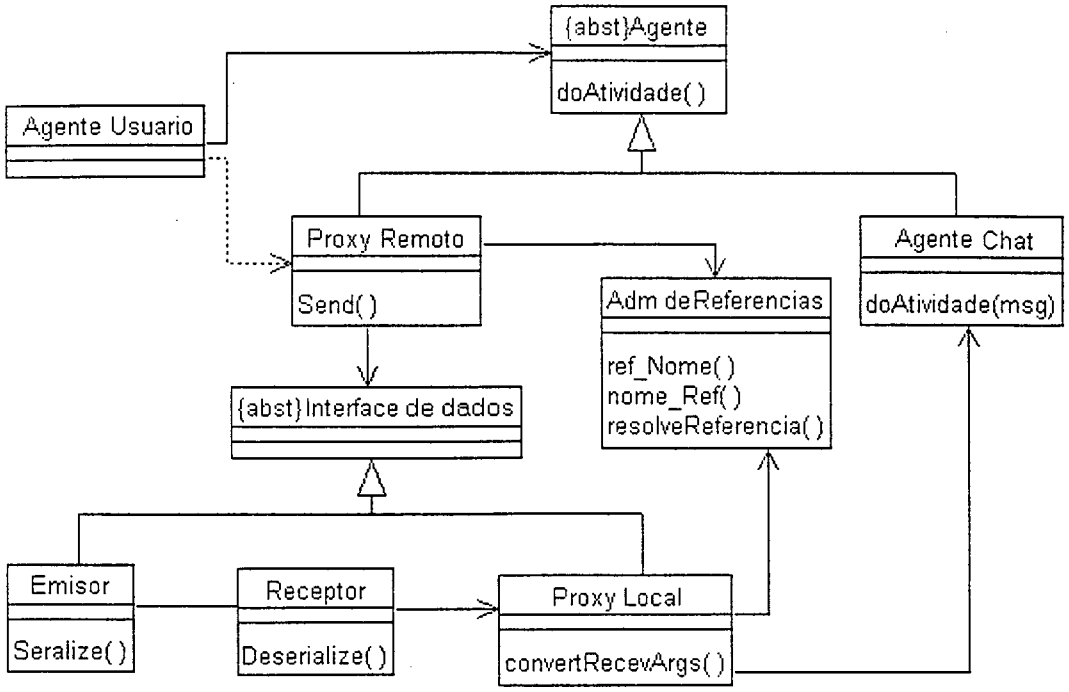


Figura 4.16. Classes que intervêm no padrão Distributed Proxy [RS97]

O SCV é funcionalmente simples, mas operacionalmente complexo. A razão é que um agente usuário não pode fazer uma referência direta ao método doAtividade() do agente chat, pois ambos são processos remotos e, portanto, a referência ao método deve ser também remota.

A solução utilizando o padrão Distributed Proxy é apresentada na figura 4.17. Esta Figura mostra as classes participantes na invocação do método *doAtividade(msg)*. Neste caso, *msg* é uma mensagem contendo o *identificador* da atividade e os argumentos correspondentes. A classe *Adm de Referências* não está contemplada devido ao fato que os agentes só têm um método acessível remotamente, o método *doAtividade()* e a classe *Interface de dados* é omitida, pois existe apenas um tipo de dados a ser enviado.

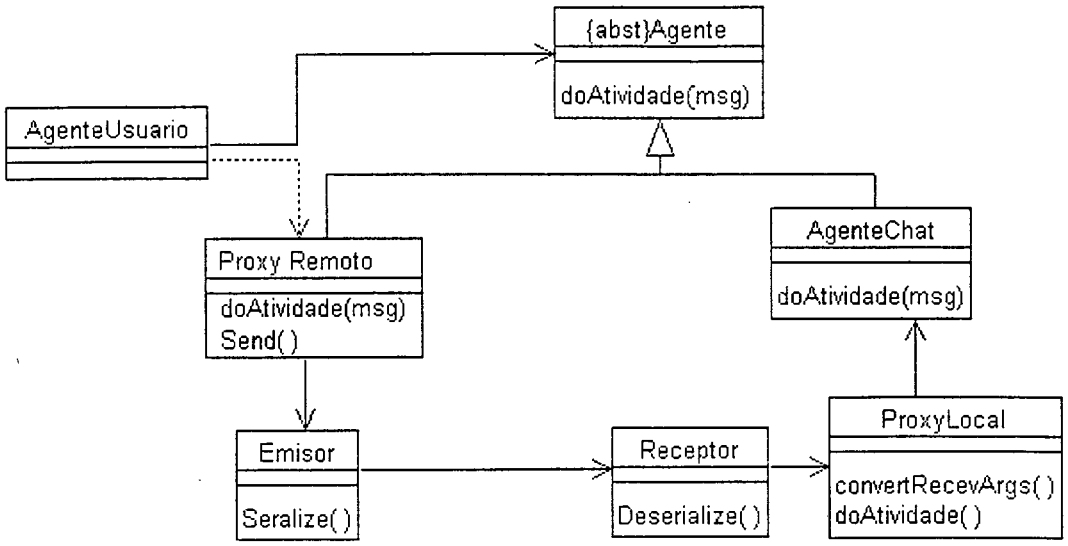


Figura 4.17. As classes participantes em uma comunicação assíncrona de A para B (cf. Figura 4.15) utilizando o padrão Distributed Proxy.

O código abaixo realiza as classes participantes.

```
public class ProxyLocal
{
            protected AgenteConcreto agente;
            protected Receptor receptor;

            public ProxyLocal(String idAgente,int port)
            {     super(idAgente);
                  agente = AgenteConcreto.crear(idAgente);
                  receptor = new Receptor("Proxy Local",port);
                  receptor.setAgente(this);
                  receptor.start();         }

            public void convertRecevArgs (Mensagem msg)
            {     agente.doAtividade(msg);        }
}
```

```
public class ProxyRemoto extends Agente
{
            private Emisor emisor;

            public ProxyRemoto(String idAgente,Endereco destino)
            {     super(idAgente);
                  emisor = new Emisor("Proxy Remoto",destino);        }

            public void doAtividade(Mensagem msg)
            {     emisor.send(msg);          }

            public Atividade getAtividade(String Atividade)
            {return null;}      }



public class Emisor
{     private String id;
      private Endereco destino;

      public  Emisor(String id, Endereco destino)
      {     this.id=id;
            this.destino = destino;    }

      public void send(Mensagem msg, Endereco destino)
      {
           try
           { Socket sock = new Socket(destino.getHost(), destino.getPort());
             OutputStream o = sock.getOutputStream();
             ObjectOutput sOut = new ObjectOutputStream(o);
             sOut.writeObject(msg);
             sOut.flush();
             sOut.close();                }
        catch (Exception e)
        {    System.out.println(e.getMessage());
             System.out.println("Error no Emisor : "+id);    }
      }
}


import java.io.*;
import java.net.*;
import java.util.*;

public class Receptor extends Thread
{     private String id;
      private ServerSocket servSock;
      private Mensagem msg;
      private Agente  agente;
      private int portLocal = 0;

      public Receptor(String id, int port)
      {     this.id=id;
            try
            {      servSock = new ServerSocket(port);
                   portLocal=servSock.getLocalPort();          }
            catch (IOException e)
            {  System.out.println(e.getMessage());             }
      }
```

```
        public void setAgente(Agente agente)
        {       this.agente = agente;}

        public void run()
        {       try
                {       while (true)
                        {       Socket sock = servSock.accept();
                                InputStream in = sock.getInputStream();
                                ObjectInput sIn = new ObjectInputStream(in);
                                msg = (Mensagem) sIn.readObject();
                                agente.doAtividade(msg);
                                sIn.close();                    }
                }
                catch (Exception e)
                {       System.out.println(e.getMessage());
                        System.out.println("Error   no   Receptor   id:      "+id);
        }
        }
}
```

As classes *Emissor* e *Receptor* correspondem à comunicação física. As classes tipo proxy correspondem a uma invocação lógica de métodos entre agentes. A invocação de métodos entre a classe *agenteUsuario* e o *agenteChat* é funcional. É importante considerar que as mensagens (que em si contêm objetos) são enviadas na forma de dados. Os canais físicos suportam dados na forma de seqüências de bytes. Logo é preciso converter a mensagem em uma seqüência de bytes antes de enviá-la.

A comunicação entre os agentes no SCV é assíncrona devido a própria natureza da aplicação. A comunicação remota entre os agentes é feita por meio de passagem de mensagens. Cada mensagem é representada por um par (id, arg) onde *id* é o identificador de algum tipo de mensagem correspondente a alguma classe de serviço e *arg* são os argumentos necessários para completar o serviço. O código da classe Mensagem é mostrado abaixo. Podemos observar que a quantidade de argumentos na criação de uma mensagem é variável e pode conter até 4 argumentos do tipo Object.

```
import java.util.Vector;

public  class Mensagem  implements java.io.Serializable
{       public String id;
        public Vector argList;

        public  Mensagem()
        {       id = null;
                argList = new Vector();   }
```

```
public  Mensagem(String id)
{     id = id;
argList = new Vector();          }
public  Mensagem(String id, Object arg1 )
{      id = id;
       argList = new Vector();
       addArg(arg1);       }

public  Mensagem(String id, Object arg1, Object arg2 )
{      id= id;
       argList = new Vector();
       addArg(arg1);
       addArg(arg2);       }

public  Mensagem(String id, Object arg1, Object arg2, Object arg3  )
{      id = id;
       argList = new Vector();
       addArg(arg1);
       addArg(arg2);
       addArg(arg3);       }

public  Mensagem(String id, Object arg1, Object arg2, Object arg3,Object
arg4  )
{      id = id;
       argList = new Vector();
       addArg(arg1);
       addArg(arg2);
       addArg(arg3);
       addArg(arg4);       }

public  String getId()
{      return id;          }

public void addArg(Object arg)
{ argList.addElement(arg);       }

public Object getArg(int i)
{return argList.elementAt(i);    }
}
```

4.5.5. Atualizações remotas

Em relação às atualizações das salas remotas, o *agente chat* deve atualizar o estado da sala toda vez que acontece um novo evento. Assim, cada *agente usuário* possuirá os dados atualizados da sala remota que participa. Por exemplo, no caso da troca de sala de um *agente usuário*, este envia uma mensagem com tal petição ao *agente chat* que por

sua vez executa a atividade correspondente sobre as duas salas. Os eventos que são gerados por cada transação devem ser propagados aos participantes de ambas as salas. Esta nova função do *agente chat* provocaria uma modificação na estrutura dos agentes e, por conseqüência, o aumento da complexidade do projeto. Uma proposta para o problema de propagação de eventos é o padrão Observer [GHJV95]. Nesse caso, as salas são observadas pelos *agentes usuários*. O diagrama de classes correspondente é mostrado na Figura 4.18.

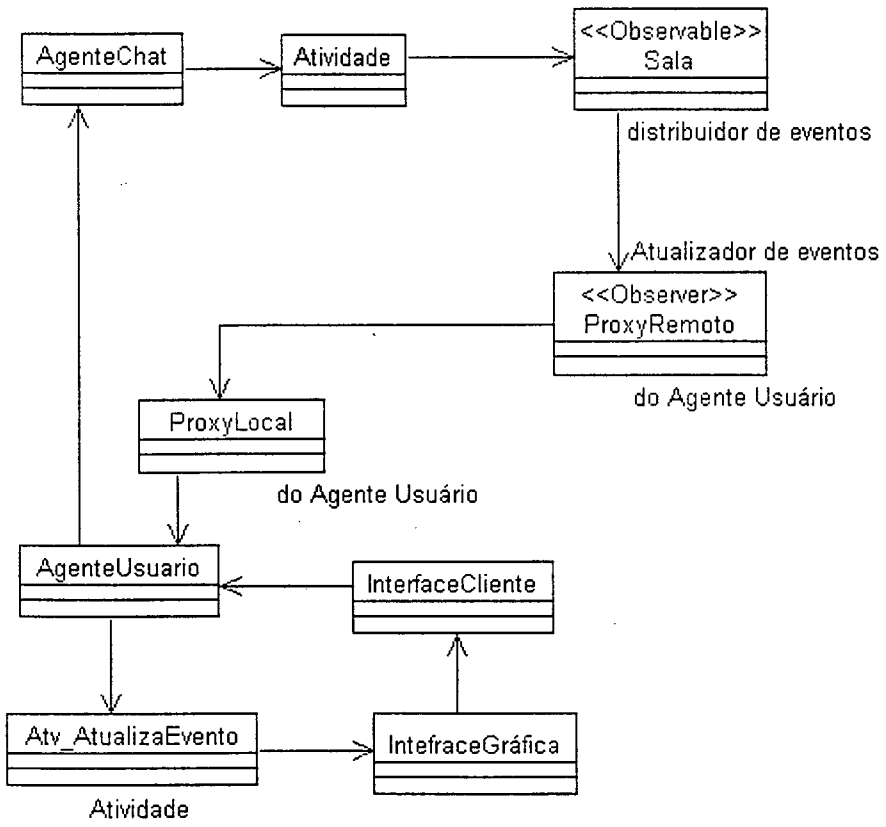


Figura 4.18 Utilização do padrão Observer para o tratamento de eventos e da interface gráfica.

O padrão Observer facilita o tratamento de eventos e alivia o *agente chat* dessa tarefa que deve entretanto notificar os eventos ao objeto *Proxy Remoto* do *agente usuário*.

Devemos também utilizar o padrão Distributed Proxy para o *agente usuário* de forma similar ao do *agente chat*. O código abaixo mostra como é implementado as classes *Sala* e *ProxyRemoto*.

```
import java.util.Observable;
import java.util.Vector;

public  class Sala   extends Observable
{      private String id;
       private String texto;
       private Vector usuarios;
       private Vector salas;
       Mensagem msgNotify; //mensage para enviar actualizacoes as classes
observers

       public  Sala(String id)
       {       this.id = id;
               texto = null;
               usuarios = new Vector();
               salas = new Vector();      }

       public boolean isVazia()
       {       return ( usuarios.isEmpty());    }

       public void atualizaSala(Vector listaSalas)
       {       this.salas = listaSalas;
               msgNotify = new Mensagem("atulizarEvento",
                                "atualizarSala",listaSalas);
               setChanged ();
               notifyObservers (msgNotify);     }

       public void addTexto(String texto)
       {       this.texto=texto;
               msgNotify = new Mensagem("atulizarEvento","addTexto",texto);
               setChanged ();
               notifyObservers (msgNotify);      }

       public void entrar(String idAgente)
        {      if(usuarios.indexOf(idAgente) == -1)
usuarios.addElement(idAgente);
               msgNotify = new Mensagem("atulizarEvento","entrar",
                                        usuarios,salas,this.getId());
              setChanged ();
               notifyObservers (msgNotify);
       }

       public void salir(String idAgente)
       {       usuarios.removeElement(idAgente);
               msgNotify = new Mensagem("atulizarEvento","salir",
                                                usuarios,this.getId());
               setChanged ();
               notifyObservers (msgNotify);     }
}
```

```
import java.util.Observer;
import java.util.Observable;

public class ProxyRemoto extends Agente implements Observer
{     private Emisor emisor;
      private Endereco endercoRemoto;

      public ProxyRemoto(String idAgente,Endereco destino)
      {     super(idAgente);
            emisor = new Emisor("Proxy Remoto",destino);
            }

            public void doAtividade(Mensagem msg)
            {     emisor.send(msg);
            }

      public void update(Observable Obs, Object Obj)
      {     doAtividade((Mensagem)Obj);
      }

      public Atividade getAtividade(String Atividade)
            {return null;}
}

public class Atv_IniciarSesion extends Atividade
{
      private static Atv_IniciarSesion Singleton = null;

      public static Atv_IniciarSesion crear()
      {     if(Singleton == null)
            Singleton = new Atv_IniciarSesion();
            return Singleton;   }

      private Atv_IniciarSesion()
      {   super("Atv_IniciarSesion"); }

      public void Execute(Mensagem Msg)
      {     String idAgente   = (String) Msg.getArg(0);
            Endereco endereco = (Endereco)Msg.getArg(1);

            ConjuntoSalas salas = ConjuntoSalas.crear();
                  ConjuntoSessoes sessoes = ConjuntoSessoes.crear();
            Agente agente = new Agente(idAgente,endereco);
            Sala sala = salas.getSalaPrincipal();

            sala.addObserver(agente);
            sala.entrar(agente.getId());
                  Sessao sessao = new Sessao();
            sessao.setAgente(agente);
            sessao.setSala(sala);
                  sessoes.addSessao(idAgente,sessao);
      }
}


public class Atv_TrocarSala extends Atividade
{
      private static Atv_TrocarSala Singleton = null;

      public static Atv_TrocarSala crear(String id)
      {     if(Singleton == null)
            Singleton = new Atv_TrocarSala(id);
```

```
            return Singleton;          }

        private Atv_TrocarSala(String id)
        {   super(id);      }

        public void Execute(Mensagem Msg)
        {       ConjuntoSalas salas      = ConjuntoSalas.crear();
                ConjuntoSessoes sessoes   = ConjuntoSessoes.crear();

                String idAgente     = (String) Msg.getArg(0);
                 String idSalaNova    = (String) Msg.getArg(1);

                Sessao sessao  = sessoes.getSessao(idAgente);
                ProxyRemoto agente = (ProxyRemoto)sessao.getAgente();
                Sala salaAtual = sessao.getSala();
                Sala salaNova  = salas.getSala(idSalaNova);

                salaAtual.salir(idAgente);
                salaAtual.deleteObserver(agente);
                salaNova.addObserver(agente);
                salaNova.entrar(idAgente);

                sessao.setSala(salaNova);
          }
}
```

A Figura 4.19 mostra os diferentes aspectos de design vistos nas seções anteriores. Os eventos que são gerados pelas transações remotas são propagados aos correspondentes agentes usuários utilizando o padrão Observer. O padrão Distributed Proxy intervém principalmente nas transações e nos eventos.

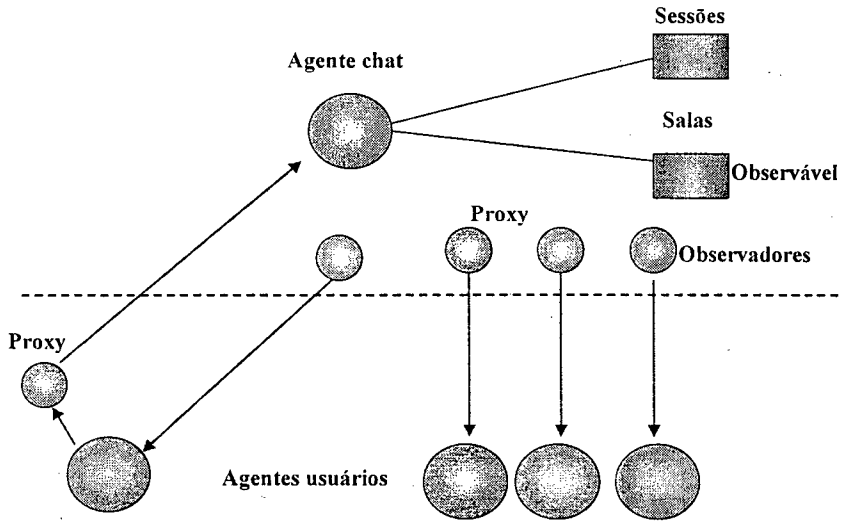


Figura 4.19 Esquematização do SCV.

4.5.6. Composição de Agentes

Finalmente, os agentes são configurados como *agente usuário* e como *agente chat*. Para cada um é criado um nome e um endereço na rede, os seus proxy local e proxy remoto e uma interface gráfica para o agente chat. Os códigos respectivos são apresentados a seguir:

```
class Cliente
{
   public static void main(String[] args)
```

```
    {   // Execução : java Cliente [nome] [port local] [host chat]
        String  idAgenteLocal     =  args[0];
        int  porCliente           =  new Integer(args[1]).intValue();
        String hostCliente        = "localHost";
        int porCliente            =  new Integer(args[1]).intValue();
        String  hostChat          =  args[2];
        int    portChat           =  8100;
        Endereco enderecoChat     = new Endereco(hostChat,portChat);
        Endereco enderecoCliente  = new Endereco(hostCliente,porCliente);
        AgenteConcreto  cliente   =  AgenteConcreto.crear(idAgenteLocal);
        cliente.setEnderecoLocal(enderecoCliente);
        ProxyLocal   proxy        =  new ProxyLocal(idAgenteLocal,porCliente);
        proxy.agente              =  cliente ;
        ProxyRemoto     proxyChat =  new ProxyRemoto("proxyChat",enderecoChat);
        cliente.agenteRemoto      = proxyChat;
        AdaptadorGUIAgente adaptador    = new AdaptadorGUIAgente();
        adaptador.agente          = cliente;
        GUIUsuario InterfaseGrafica     = GUIUsuario.crear(idAgenteLocal);
        InterfaseGrafica.agente   = adaptador;
        InterfaseGrafica.pack();
        InterfaseGrafica.show();
        InterfaseGrafica.addNuevoMensage(" bien venido : "+idAgenteLocal);
     }
}

class Chat
{
   public static void main(String[] args)
   {   String  idAgenteLocal     =  "chat";
       String  hostChat          =  "localHost";
       int    portChat           =  8100;
       Endereco enderecoChat     =  new Endereco(hostChat,portChat);
       AgenteConcreto  chat      =  AgenteConcreto.crear(idAgenteLocal);
       chat.setEnderecoLocal(enderecoChat);

       ProxyLocal  proxy         =  new ProxyLocal(idAgenteLocal,portChat);
       proxy.agente              =  chat ;
      System.out.println("Server pronto");
     }
}
```

4.6. Conclusões

Neste capítulo, foi desenvolvido o sistema de conversação virtual, uma aplicação distribuída representativa de uma classe de aplicações distribuídas: os sistemas cooperativos. Embora seja uma aplicação simples na sua abordagem, tentou-se aqui generalizá-la com a finalidade de ser utilizada como "campo de prova" para uma metodologia de desenvolvimento de sistemas cooperativos centrada em padrões de design.

Na fase de análise, definiu-se três classes principais: o agente usuário, o agente chat e a sala com suas correspondentes responsabilidades. Um protótipo da interface gráfica foi proposto para a visualização das funcionalidades da aplicação. As relações entre as classes envolvidas na concepção do sistema foi definida num diagrama de classes e serviu de base para o projeto de toda a aplicação.

Na fase de projeto, foram tratados quatro aspectos: a estrutura da aplicação, os serviços fornecidos, as petições remotas de serviços e as atualizações remotas. Em cada um desses aspectos, os padrões de design foram de grande ajuda no momento de enfrentar os problemas de projeto próprios desse tipo de aplicação. Alguns deles são comentados a seguir:

Problemas	**Padrão utilizado**
Deve existir só um agente chat Deve existir só um agente usuário representando cada usuário do sistema	Singleton
Uma mensagem recebida deve ser traduzida para uma atividade	Abstract Factory
Desacoplamento da comunicação física da comunicação lógica entre os agentes	Proxy Distributed Proxy

Problemas	Padrão utilizado
Desacoplamento entre: Agente – Objeto sujeito Agente – Atividade Atividade – Objeto sujeito	Combinação dos padrões Template e Objectifier
Propagação de eventos e troca de textos entre agentes usuários	Observer
Desacoplamento entre o agente usuário e a interface gráfica	Adapter

Em cada um desses problemas os padrões de design proporcionaram soluções confiáveis, facilitando a construção da aplicação e proporcionando uma estrutura válida que pode ser reutilizada.

O SCV foi realizado utilizando a plataforma Java de Sun Microsystems.Utilizou-se o suporte de comunicação via "socket" disponível em Java em detrimento de outras tecnologias de comunicação mais sofisticadas. Embora não evidente, tecnologias tais como CORBA ou RMI, empregadas com sucesso em redes locais, possuem certos inconvenientes quando utilizadas sobre o ambiente Internet. Uma rede local possui propriedades que a WEB não pode oferecer. Uma discussão detalhada a esse respeito pode ser encontrada em [CAR99]. Além disso, a comunicação via socket permite a construção de estruturas de comunicação mais flexíveis e mais adaptáveis às características intrínsecas de aplicações cooperativas em ambiente aberto.

CAPÍTULO 5

REUTILIZAÇÃO

5.1. Introdução

A utilização dos padrões de design no projeto apresentado no capítulo precedente teve como objetivo principal a reutilização. Neste capítulo, proporemos algumas modificações no projeto do SCV com objetivo de avaliar o seu grau de reutilização. As modificações serão tanto a nível funcional quanto a nível estrutural. No funcional, adicionaremos duas funcionalidades ao SCV:

- Liderança de sala
  Qualquer usuário da sala pode ser o líder.

- Cancelamento de usuário
  O líder de uma sala pode cancelar a participação de qualquer um de seus usuários.

No nível estrutural, a reutilização é colocada mais fortemente a prova. Passaremos de uma estrutura baseada em agentes fixos, o caso apresentado no capítulo precedente, a uma estrutura baseada em agentes móveis.

A avaliação dos resultados da modificação é feita empiricamente considerando o grau

de reaproveitamento do código. Mesmo assim, em ambos os casos, foi obtido uma avaliação bastante positiva da utilização dos padrões de design no projeto do SCV.

O restante deste capítulo é dedicado à apresentação das modificações e à análise dos respectivos resultados. Começaremos pelas modificações funcionais.

5.2. Incorporando novas funcionalidades

São duas as funcionalidades estendidas: liderança de sala e cancelamento de usuário. A Figura 5.1 mostra o novo fluxo de atividades. Qualquer evento associado a entrada ou saída de um usuário pode provocar a definição de um novo líder de sala. Por facilidade e sem fugir do objetivo principal deste trabalho, o novo líder é definido por circunstância. A liderança é sempre definida por antigüidade. Quando um novo líder é definido para uma sala, todos os seus usuários são notificados através da atividade *AtualizarLíder()*. O cancelamento de um usuário pelo líder gera uma mensagem para que seja executada a atividade *CancelarUsuario()*. O código relativo a essas novas funcionalidades são apresentados a seguir:

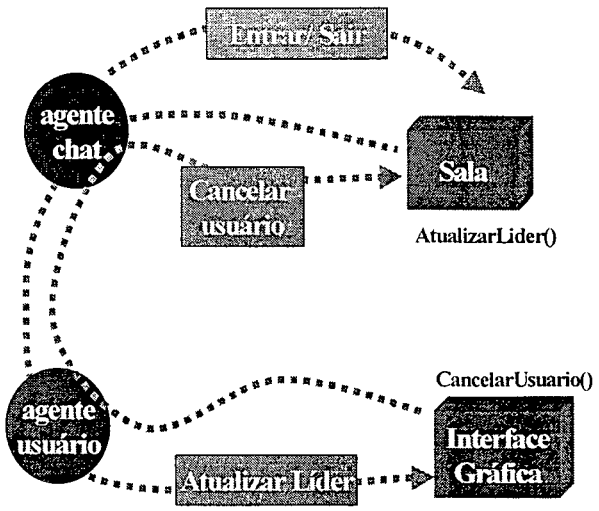


Figura 5.1 Novos fluxos de atividades.

```
// Atividade que atuliza a designação do novo administrador da sala.
//
import java.util.Vector;
public class Atv_AtualizarEvento extends Atividade
{
      :
      public void Execute(Mensagem msg)
      {
            GUIUsuario gui      = GUIUsuario.crear("");
            gui.pack();
            gui.show();
            String operacao = (String)msg.getArg(0);
      :
      if (operacao.equals("novoLider"))
      {      String usuarioLider = (String)msg.getArg(1);
            gui.actualizaUsuarioLider(usuarioLider);
       }
      }
}



// Na classe Sala é acrescentada um método: AtualizarLider()
//
import java.util.*;

public  class Sala   extends Observable
{     :
      public void entrar(String idAgente)
      :
      public void salir(String idAgente)
      :

      public void AtualizarLider()
      {     if(isSalaVazia())
            {      usuarioLider=null;
                  return;
            }

            if(!(usuarios.contains(usuarioLider)))
            {     usuarioLider=(String)(usuarios.firstElement());
                  //Notifica salida de un usuario
                  msgNotify = new Mensagem("atulizarEvento",
                  "nuevoLider",usuarioLider,this.getId());
                  setChanged ();
                  notifyObservers (msgNotify);
            }
      }

}
```

```
// Uma nova atividade, a que elimina um usuário da sala,
//é acrescentada.
public class Atv_CancelarUsuario extends Atividade
{
      private static Atv_CancelarUsuario Singleton = null;

      public static Atv_CancelarUsuario crear()
      {     if(Singleton == null)
            Singleton = new Atv_CancelarUsuario();
            return Singleton;
      }
      private Atv_CancelarUsuario()
      {   super("Atv_EliminarUsuario");
      }

      public void Execute(Mensagem Msg)
      {       System.out.println("Ejectando :"+ this.getId());

            String idAgente    = (String) Msg.getArg(0);
            String idAgenteParaEliminar    = (String) Msg.getArg(1);

            ConjuntoSalas salas = ConjuntoSalas.crear();
              ConjuntoSessoes sessoes = ConjuntoSessoes.crear();

            Sessao sessao = sessoes.eliminarSessao(idAgenteParaEliminar);

            ProxyRemoto agente = (ProxyRemoto) sessao.getAgente();
            Sala sala = sessao.getSala();

            sala.deleteObserver(agente);
            sala.salir(agente.getId());
      }
}
```

As classes agentes não foram afetadas pelas novas funcionalidades. As classes ***Sala*** e ***InterfaceGrafica*** sofreram pequenas modificações. Duas novas classes atividades foram criadas: uma para a notificação do novo líder e outra para o cancelamento de um usuário. Podemos ver claramente que o impacto provocado pela inclusão dessas funcionalidades foi mínimo. A razão é que as classes participantes (agentes, atividades e objetos em geral) são fracamente acopladas. Resultado que é atribuído em grande parte a utilização dos padrões de design no desenvolvimento do SCV.

5.3. Modificando a estrutura de interação

Conceitualmente, a estrutura de interação do projeto SCV do capítulo anterior é estática. Isto quer dizer que os objetos e agentes interagem sempre em um mesmo ambiente. No

caso em questão, o ambiente é definido pelo servidor do SCV. A funcionalidade "trocar de sala" é efetuada por simples manipulação de parâmetros. As salas são simples referências a agentes usuários. Existe portanto um acoplamento relativamente forte entre os agentes usuários e o agente chat na realização deste tipo de funcionalidade. Era de se esperar entretanto que o "movimento" dos agentes usuários entre as salas fosse feito de forma autônoma em co-responsabilidade apenas com as salas envolvidas. Ao agente chat apenas a responsabilidade da localização dos agentes usuários, feita por simples notificação. Esta é a modificação estrutural pretendida.

É importante observar que a noção de mobilidade dos agentes usuários é conceitual. Os agentes permanecem no mesmo ambiente físico ("website"), mas movem-se de um ambiente de interação (sala) para outro. Supõe-se que o leitor esteja suficientemente esclarecido sobre este tipo de conceituação. Caso contrário, recomendamos dois excelentes trabalhos sobre esse assunto: o Calculo-Pi [MIL92] e o Cálculo de Ambientes [CAR97].

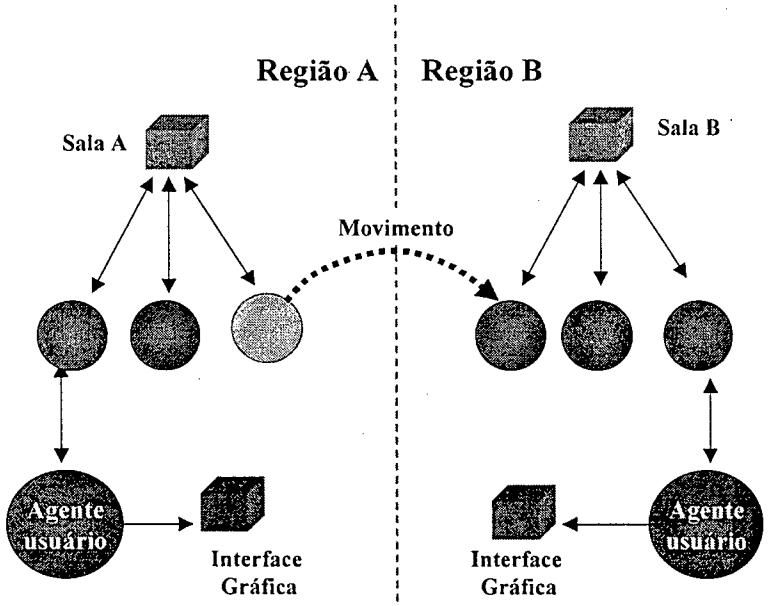


Figura 5.2 O movimento de agentes móveis da região A para região B.

A Figura 5.2 apresenta uma esquematização da estrutura de interação proposta. Nesta estrutura, o agente chat continua sendo fixo. Apenas os agentes usuários possuem mobilidade. Como não existe mobilidade física, por facilidade, optou-se por uma mobilidade via mudança de observação. Para tanto, utilizou-se o padrão Observer tendo a sala como objeto observável e os agentes usuários como observadores.

```
mport java.net.*;
import java.lang.*;
import java.util.*;
class  AgenteMovel extends  Agente implements Observer
{
      public Agente agenteChat = null;
      public Agente agenteUsuario= null;

        public AgenteMovel(String idAgente)
        {    super(idAgente);
        }

        public void doAtividade(Mensagem msg)
        {    agenteChat.doAtividade(msg);

        }


      public void update(Observable Obs, Object Obj)
      {      agenteUsuario.doAtividade((Mensagem)Obj);
      }

      public Atividade getAtividade(String idAtividade )
      {      return null;
      }

}


//Classe que cria o agente movel
//
public class Atv_IniciarSessao extends Atividade
{     private static Atv_IniciarSessao Singleton = null;
      public static Atv_ IniciarSessao criar()
      {      if(Singleton == null)
             Singleton = new Atv_ IniciarSessao ();
             return Singleton;
      }
      private Atv_ IniciarSessao ()
      {   super("Atv_ IniciarSessao ");
      }

      public void Execute(Mensagem Msg)
      {      System.out.println("Ejectando :"+ this.getId());
             String idAgente    = (String) Msg.getArg(0);
             Endereco endereco = (Endereco)Msg.getArg(1);
             ConjuntoSalas salas = ConjuntoSalas.criar();
             ConjuntoSessoes sessoes = ConjuntoSessoes.criar();

             AgenteMovel agente = (AgenteMovel)
```

```
            CriarAgente(idAgente,endereco);

            Sala sala = salas.getSalaPrincipal();
            sala.addObserver(agente);
            sala.entrar(agente.getId());

            Sessao sessao = new Sessao();
            sessao.setAgente(agente);
            sessao.setSala(sala);

            sessoes.addSessao(idAgente,sessao);
        }

        private Agente CriarAgente(String idAgente,Endereco
                                                    enderecoAgenteRemotoFixo)
        {
         FactoryEndereco factoryEndereco = FactoryEndereco.criar();
         Endereco enderecoAgenteMovel    =
                                  factoryEndereco.crearEndereco();
         Endereco enderecoAgenteFixo     = enderecoAgenteRemotoFixo;

         AgenteMovel agenteMovel         =  new AgenteMovel(idAgente);
         agenteMovel.setEnderecoLocal(enderecoAgenteMovel);

        proxyLocal   proxy = new ProxyLocal(idAgente,enderecoAgenteMovel);
        proxy.agente      =  agenteMovel ;
        proxyRemoto  proxyUsuario  =  new
                     proxyRemoto(idAgente,enderecoAgenteRemotoFixo);
         agenteMovel.agenteUsuario  =  proxyUsuario;

         AgenteConcreto chat = AgenteConcreto.crear("Agente Chat");
         agenteMovel.agenteChat = chat;

         return agenteMovel;
        }

}
```

```
//A classe que elimina o agente movel
//
public class Atv_TrocarSala extends Atividade
{
	private static Atv_TrocarSala Singleton = null;

	public static Atv_TrocarSala crear(String id)
	{	if(Singleton == null)
		Singleton = new Atv_TrocarSala(id);
		return Singleton;
	}
	private Atv_TrocarSala(String id)
	{   super(id);
	}

	public void Execute(Mensagem Msg)
	{
		ConjuntoSalas salas     = ConjuntoSalas.criar();
		ConjuntoSessoes sessoes = ConjuntoSessoes.criar();

		String idAgente     = (String) Msg.getArg(0);
		  String idSalaNova = (String) Msg.getArg(1);

		Sessao sessao  = sessoes.getSessao(idAgente);
		AgenteMovel agente = (AgenteMovel)sessao.getAgente();
		Sala salaAtual = sessao.getSala();
		Sala salaNova  = salas.getSala(idSalaNova);

		salaAtual.salir(idAgente);
		salaAtual.deleteObserver(agente);
		salaNova.addObserver(agente);
		salaNova.entrar(idAgente);

		sessao.setSala(salaNova);
	  }
}
```

```
import java.net.*;
import java.lang.*;
import java.util.*;
class  AgenteUsuario extends Agente
{
        private static AgenteUsuario Singleton = null;

        public Agente agenteChat=null;
        public Agente agenteMovel=null;

        public static AgenteUsuario crear(String idAgente)
      {     if(Singleton == null)
            Singleton = new AgenteUsuario(idAgente);
            return Singleton;
      }

        private AgenteUsuario(String idAgente)
        {   super(idAgente);
        }

        public void doAtividadeChat(Mensagem msg)
      {     agenteChat.doAtividade(msg);
      }

      public void doAtividadeMovil(Mensagem msg)
      {     agenteMovel.doAtividade(msg);
      }

      public Atividade getAtividade(String idAtividade )
      {     FactoryAtividade factActividade = FactoryAtividade.crear();
            return(factActividade.crearAtividade(idAtividade));
      }

}
```

Esses são os códigos mais importantes do ponto de vista das modificações efetuadas. O agente chat permanece intacto, mas o agente usuário, agora agente usuário local, foi modificado. Foi necessário estabelecer uma ligação entre ele e o agente usuário móvel.

5.4. Conclusões

Neste capítulo, foi abordado o problema da reutilização do projeto desenvolvido no capítulo anterior tanto nos aspectos funcionais quanto nos estruturais.

Foi introduzida duas novas funcionalidades:

- Escolher um líder entre os participantes da sala.
- O usuário líder pode cancelar um dos usuários da sala.

O impacto de introduzir as novas funcionalidades foi mínima devido ao fraco acoplamento entre as classes envolvidas.

O segundo tipo de modificação introduziu a noção de mobilidade para os agentes usuários. Cada sala agrega agora um conjunto de agentes usuários móveis que representam o usuário na sala. Esses agentes, diferente dos proxies associados aos agentes usuários locais, são autônomos. Podem executar uma atividade e fazer petições a qualquer agente residindo na mesma sala.

A modificação na estrutura de interação entre os agentes causou pouco impacto no projeto. O agente chat e a estrutura de comunicação permaneceram intactos. A introdução da figura do agente usuário móvel gerou poucas modificações no design do agente usuário local. Esse resultado foi obtido graças à utilização de padrões de design que orientou o projeto do SCV para um alto grau de reutilização.

CAPÍTULO 6

CONCLUSÃO

Este trabalho abordou a problemática de projeto de sistemas cooperativos utilizando padrões de design como uma ferramenta essencial na busca de soluções flexíveis e reutilizáveis.

Os padrões de design foram apresentados no Capítulo 2. Alguns dos mais conhecidos foram estudados em detalhe. Para validar a metodologia, foi desenvolvido um sistema de conversação virtual (SCV) sobre a Internet utilizando a plataforma Java da Sun Microsystems. O modelo de computação distribuída adotado foi o de passagem de mensagens entre agentes fixos. As mensagens são constituídas de um identificador do tipo da mensagem e de argumentos. Os agentes sendo considerados processos independentes com uma funcionalidade bem definida. O capítulo 5 apresentou dois aspectos de reutilização do projeto SCV: extensão de suas funcionalidades e modificação de sua estrutura de interação. A linguagem UML foi utilizada para a descrição dos aspectos de concepção, estruturais e comportamentais do projeto.

Em relação aos resultados observados na fase de concepção, realização e modificações do SCV podemos destacar os seguintes pontos:

- Entre os aspectos mais relevantes, destacam-se a estrutura da aplicação, os serviços fornecidos, as petições remotas de serviços e as atualizações remotas.

- Os agentes foram considerados somente como processos comunicantes sem distingüir os agentes usuários do agente chat. Dessa forma, todos os agentes tem a mesma estrutura básica e os mesmos componentes para receber e enviar mensagens.

- Visualizar os processos comunicantes como agentes facilita a construção de sistemas com estrutura adaptável, pois neste caso os agentes podem modificar a sua estrutura para se adaptarem a novos tipos de mensagens.

- A partir de uma mesma estrutura de agente é possível gerar diferentes tipos de agentes através do acréscimo de funcionalidades. Neste caso, utilizou-se o padrão Adapter [GHJV95] para permitir que diferentes componentes, mesmo com interfaces discordantes, possam ser associados.

- O padrão Template em conjunto com o Objetifier facilitou o projeto da estrutura básica dos agentes, permitindo que eles pudessem ser modelizados como processos cooperantes que realizam atividades abstratas. Isto tornou o projeto mais flexível e reutilizável, pois os agentes e atividades são fracamente acoplados.

- O padrão Singleton cumpre um papel importante em grande parte do projeto uma vez que permite que os objetos possam ser acessado globalmente e instanciados unicamente quando são utilizados. No projeto proposto, cada atividade a ser realizada pelo agente é do tipo Singleton.

- Um dos aspectos de maior complexidade é a invocação de métodos de objetos remotos. Isto é simples em sistema tais como CORBA ou RMI de Java, pois já provêm a localização dos objetos de forma transparente, ocultando a complexidade da comunicação envolvida. Na proposta que foi desenvolvida utiliza-se o padrão Distributed Proxy para prover esta funcionalidade. A solução consiste em considerar a comunicação em três camadas: uma física que é a comunicação a nível de transmissão de bytes; logo acima, uma camada que trata da comunicação lógica transformando uma chamada a um objeto remoto em uma mensagem a ser transmitida; e, no topo, uma camada funcional que permite tratar o acesso aos métodos remotos como se fossem objetos locais.

- Para obter os resultados desejados é necessário que o projetista conheça e domine

um conjunto de padrões relacionados.

- Na problemática de projeto de sistemas cooperativos, os padrões de design podem ser de grande utilidade para resolver problemas complexos com soluções já conhecidas, estudadas e consolidadas.

- A utilização de padrões de design no projeto pode garantir flexibilidade, na hora de incorporar novas funcionalidades, e um alto grau de reutilização de projeto mesmo quando a estrutura de interação é alterada.

- A linguagem Java mostrou-se bastante adequada para o desenvolvimento de aplicações cooperativas em ambiente Internet. A inclusão de alguns padrões de design possibilitou uma facilidade maior no desenvolvimento do trabalho.

Entre as várias perspectivas de continuação deste trabalho podemos destacar as seguintes:

- O desenvolvimento de uma biblioteca de padrões de design para a linguagem Java de forma similar ao que acontece com o padrão Observer.

- A partir de uma coleção de padrões de design podemos imaginar um conjunto de regras de aplicação. Esse raciocínio nos leva a imaginar uma linguagem de padrões de design que associada a uma determinada arquitetura de software definiria toda uma metodologia de projeto orientada a padrões de design.

- Na arquitetura de software utilizada no projeto do SCV, os agentes são estáticos. A modificação apresentada no Capítulo 5 fornece apenas uma mobilidade conceitual (há um único ambiente onde a computação evolui) e restringida aos agentes usuários. Uma noção de mobilidade mais abrangente e fisicamente distribuída permitiria um melhor aproveitamento do ambiente computacional disponível e um grau maior de eficiência e confiabilidade na operação do sistema.

BIBLIOGRAFIA

[BRA] Endereço com informação sobre padrões de design
http://www.enteract.com/~bradapp/docs/patterns-intro.html#KindsOfPatterns

[BT98] Bob Tarr, "Design Patterns In Java", Curso sobre padrões de design, CMSC491X Design Patterns in Java.
http://www.research.umbc.edu/~tarr/cs491/fall98/cs491.html

[CA+77] Christopher Alexander et al., *A Pattern Language*, Oxford University, 1977.

[CAR97] Luca Cardelli, Mobile Ambients, Digital Corporation, System Research Center.
http://www.luca.demon.co.uk/Bibliography.html#Abstractions for Mobile Computation

[CAR99] Automata, Languagese and Programming, 26th International Colloquium, ICALP'99 Proceedings. Leitura em Computer Science, Vol. 1644, Springer, 1999. ISBN 3-540-66224-3. pp. 10-24.
http://www.luca.demon.co.uk/Bibliography.html#Wide Area Computation

[CS95] James O. Coplien and Douglas Schmidt (eds.), Pattern Languages of Program Design, Addison-Wesley, 1995.

[DL98] Doug Lea, "Design for Open System in Java", State University of New York at Oswego, Coordination 1997.

[EL99] Elliote Rusty Harold, "Java I/O", O'REILLY, 1999.

[ELI96] Elizabeth A. Kendall, Margaret T. Malkou. Computer System Engineering, Royal Melbourne Institute of Technology, apresentado em PLoP 1996.

[JF98] Jim Farley, "*Java Distributed Computting* ", O'REILLY, 1998

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[IV99] Ivor Horton, "Beginning Java 2", Wrox Press Ltda., 1999.

[JC95] James O. Coplien, "A Generative Development-Process Pattern Language", in [CS95]

[MIL92] Milner, R., J. Parrow and D. Walker, *"A Calculus of Mobile Processes"*, Part 1-2. Information and Computation, 100(1), 1-77. 1992.

[MF97] Martin Fowler , "UML DESTILLED, Applying the Standard Object Modeling Language", Addison-Wesley,1997.

[MS89] Mary Shaw, Larger Scale Systems Require Higher-Level Abstractions", in *Proceedings of Fifth International Workshop on Software Specification and Design*, IEEE Computer Society, 1989.

[RG96] Richard Gabriel, *Patterns of Software: Tales From the Software Community*, Oxford, 1996.

[RO98] Robert Orlafi Dan Harkey, *"Client/Server Programing with Java and ORBA"*, John Wiley & Sons,Inc, Second Edition, 1998

[RS97] Antônio Rito Silva, "Distributed Proxy: A Design Pattern for Distributed Object Communication", PloP97, Illinois, USA, Setembro 1997.

[PloP] *Pattern Languages of Programs*, realizadas anualmente, desde 1994, em Illinois, EUA, no mês de setembro.

[WZ97] W. Zimmer, "Relationships between Design Patterns," in Pattern Languages of Program Design.